1996

Timed data flow diagrams

Jürgen Symanzik
Iowa State University

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Timed data flow diagrams

by

Jürgen Symanzik

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY

Co-majors: Statistics; Computer Science

Major Professors: Herbert T. David and Albert L. Baker

Iowa State University

Ames, Iowa

1996

Graduate College
Iowa State University

This is to certify that the Doctoral dissertation of

Jürgen Symanzik

has met the dissertation requirements of Iowa State University

Signature was redacted for privacy.

Co-major Professor

Signature was redacted for privacy.

Co-major Professor

Signature was redacted for privacy.

For the Co-major Program

Signature was redacted for privacy.

For the Co-major Program

Signature was redacted for privacy.

For the Graduate College

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Traditional Data Flow Diagrams (DFD's) are the cornerstone of the software development methodology known as "Structured Analysis" (SA), and they are probably the most widely used specification technique in industry today. DFD's are popular because of their graphical representation and their hierarchical structure. Thus, they are well–suited for users with non–technical backgrounds and are commonly used to depict the static structure of information flow in a system. Numerous attempts to formalize DFD's have appeared in the technical literature. We focus on the Formalized Data Flow Diagrams (FDFD's) developed by Coleman, Wahls, Baker, and Leavens.

This dissertation analyzes and extends FDFD's with respect to their usefulness in specifying the qualitative and quantitative properties of real systems. Prior to this dissertation, there existed no well–founded knowledge about the computational power of FDFD's nor any formal model in FDFD's of the timing behavior of real systems.

The dissertation is organized as a collection of five independent papers. Briefly, the main results of each paper are as follows: (i) Reduced FDFD's are Turing equivalent. (ii) Stores, persistent flows, tests for empty flows, and infinite domains are not essential for FDFD's. (iii) Subclasses of FDFD's are equivalent to known subclasses of FIFO Petri Nets, immediately furnishing the decidability results for subclasses of FIFO Petri Nets to the corresponding subclasses of FDFD's. (iv) A general stochastic model of time for FDFD's (called Timed Data Flow Diagrams — TDFD's) is defined, allowing not only a description of the relative likelihoods of various execution times, but also descriptions of the possible joint firing behavior of transitions. (v) An aggregation principle can be used for an efficient stochastic analysis of periodic TDFD's with Markovian transition times.

The results in this dissertation provide a firm theoretical foundation for further advances in Computer Science and Statistics, leading to practical and expressive tools for the specification and analysis of real systems.

1 GENERAL INTRODUCTION

1.1 Problem Statement

Traditional Data Flow Diagrams (DFD's) are the cornerstone of the software development methodology known as "Structured Analysis" (SA) ([DeM78], [WM85a]), and they are probably the most widely used specification technique in industry today ([BB93]). DFD's are popular because of their graphical representation and their hierarchical structure. Thus, they are ideally suited for users with non-technical backgrounds and are commonly used to depict the static structure of information flow in a system.

Traditional DFD's consist of a set of bubbles and a set of labeled flows. Bubbles represent either processes in a concurrent system or sequential procedures and are usually drawn as circles, ovals, or boxes. Flows represent data paths and are drawn as arrows connecting the bubbles. Flows coming into a bubble are called inflows and flows leaving a bubble are called outflows. Informally, a bubble reads the information on its inflows, and produces new information on its outflows. There are two types of flows. Persistent flows are like shared variables whose values are written by the source bubble and read by the destination bubble. Consumable flows are modelled as unbounded FIFO queues with the source bubble enqueueing values on the queue and the destination bubble dequeueing values.

Even though traditional DFD's are popular, they lack formality and do not provide a rigorous definition of system functionality. Numerous attempts to formalize DFD's have appeared in the technical literature, e. g., in [DeM78], [WM85a], [WM85b], [Har87], [TP89], [You89], [Har92], and [Har96]. We focus on a computational model known as Formalized Data Flow Diagrams (FDFD's) developed by Coleman, Wahls, Baker, and Leavens in [Col91], [CB94], [WBL93], and [LWBL96].

The main intent of this dissertation is to extend the knowledge about FDFD's, with respect to interesting qualitative (e. g., deadlock, reachability, termination, finiteness, liveness) and quantitative (e. g., performance, throughput, average load of a bubble) properties of real systems. We decided to restrict the research done in this dissertation to five loosely related topics on Formalized Data Flow Diagrams and a certain model, Timed Data Flow Diagrams (TDFD's), introduced in this dissertation

for the purpose of modeling timing in FDFD's. Prior to this dissertation, there existed no well-founded knowledge about the computational power of FDFD's nor any formal model in FDFD's of the timing behavior of real systems. These are the two main aspects dicussed here. However, many questions that could not be answered in this dissertation, because of time and space limitations, deserve future consideration.

In particular, we answer the following questions about the computational power of FDFD's: Do FDFD's have the same computational power as FIFO Petri Nets (introduced in [MM81]), another specification technique for concurrent and distributed systems? If so, to which class of FIFO Petri Nets are FDFD's equivalent? Are FDFD's Turing equivalent? Finally, do features of FDFD's like persistent flows, stores, infinite domains for flow values, and the ability to test for empty flows add to the computational power of FDFD's?

Are there subclasses of FDFD's that are equivalent to other computational models? Can we solve decidability questions for FDFD's by mapping the FDFD onto other computational models?

We next address Timed (or Stochastic) Data Flow Diagrams (TDFD's or SDFD's). Our model of time allows each transition of the TDFD to depend on a different deterministic (or stochastic) firing time, conditional on the previous history of transitions and on the current state. For these general models, simulation is the only available means to analyze the associated stochastic process. However, if we restrict the type of distributions and the execution policies used, we can base our statistical analysis of the TDFD on methods developed for (semi) Markov processes or Markov chains, to mention only a few. In particular, we show how an efficient stochastic analysis can be done for periodic TDFD's with Markovian transition times.

1.2 Dissertation Organization

The main idea of this Ph.D. research is (i) to establish a framework for comparing FDFD's with other computational models and for determining those features of FDFD's that are essential and (ii) to define a time behavior for FDFD's. In (i), we compare FDFD's with Turing Machines, Program Machines, and FIFO Petri Nets. We will see that some features of FDFD's only improve the expressive power but not the computational power of FDFD's. Finally, we will be able to relate some subclasses of FDFD's to some well known subclasses of FIFO Petri Nets. i. e., Monogeneous, Linear, and Topologically Free Choice FIFO Petri Nets. This comparison of FDFD's with other computational models helps to get a more sound understanding of the relevant features of FDFD's. For example, prior to this work it was not obvious that the test for empty flows does not raise the power of FDFD's, whereas it is known that

Petri Nets with so–called inhibitor arcs (a transition may fire if a place is empty) have the computational power of Turing Machines.

In (ii) we define a general model of Timed (or Stochastic) Data Flow Diagrams (TDFD's or SDFD's) that allows many types of deterministic and stochastic time behavior in combination with different basic execution policies. Applications of TDFD's and SDFD's are given.

This dissertation is organized as a collection of five papers (Chapters 2 to 6). The papers are preceded by a general introduction (Chapter 1) and followed by a general summary (Chapter 7). All references cited in the dissertation (including the collected papers) follow the general summary. As the collected papers are self–contained, similar motivational material appears in several of the papers. However, the technical content of each paper is different, since each paper focuses on a different aspect of FDFD's and TDFD's — the first three papers on computational aspects and the last two papers on the timing behavior, respectively. A short summary (i. e., the abstract) of each of the papers follows in the next section.

1.3 Dissertation Summary

Paper 1: Formalized Data Flow Diagrams and Their Relation to Other Computational Models

One approach to the formalization of Data Flow Diagrams (DFD's) is presented by Coleman ([Col91], [CB94]) and Leavens, et al., [LWBL96]. These Formalized Data Flow Diagrams (FDFD's) can be viewed as another model of computation. This paper contains an analysis of the computational power of these FDFD's. We first consider the issue whether certain features of FDFD's affect their computational power. A Reduced Data Flow Diagram (RDFD) is an FDFD with no stores, finite domains for flow values, and no facility for testing for empty flows, but it may contain persistent flows. An RDFD without persistent flows is called a persistent flow–free Reduced Data Flow Diagram (PFF–RDFD). We show that PFF–RDFD's are Turing equivalent. The other features of FDFD's only add to the expressive power of FDFD's ([SB96b]). Therefore, any FDFD can be expressed as an PFF–RDFD. Our proof that PFF–RDFD's are Turing equivalent procedes as follows. We first show that any RDFD can be simulated by a FIFO Petri Net. We then show that any Program Machine can be simulated by an PFF–RDFD. It is known that FIFO Petri Nets and Program Machines both are Turing equivalent.

Paper 2: Non–Atomic Components of Data Flow Diagrams: Stores, Persistent Flows, and Tests for Empty Flows

It has been shown in [SB96a] that a particular subclass of Formalized Data Flow Diagrams (FDFD's) is Turing equivalent. We call this Turing equivalent subclass of FDFD's persistent flow–free Reduced Data Flow Diagrams (PFF–RDFD's). PFF–RDFD's do not contain persistent flows, reference only values whose types have finite domains, and have enabling conditions that contain no tests for empty flows. In addition, FDFD's do not contain (direct) representations of stores. This raises the question whether any of these common features of traditional Data Flow Diagrams elevates the expressive power of FDFD's, or whether the various subclasses have the same expressive power as FDFD's with these features. This paper addresses this issue of whether persistent flows, arbitrary domains, tests for empty flows or stores are essential features with respect to the expressive power of Formalized Data Flow Diagrams.

Please note that we are using a different (in comparison to Papers 1, 3, 4, and 5) but equivalent notation for FDFD's in this paper. This is intentional since this paper addresses a more practically oriented readership and therefore uses the somewhat more intuitively understandable notation while the other four papers focus on theoretical aspects where the operational notation is more appropriate.

Paper 3: Subclasses of Formalized Data Flow Diagrams: Monogeneous, Linear, and Topologically Free Choice RDFD's

Formalized Data Flow Diagrams (FDFD's) and, especially, Reduced Data Flow Diagrams (RDFD's) are Turing equivalent ([SB96a]). Therefore, no decidability problem can be solved for FDFD's in general. However, it is possible to define subclasses of FDFD's for which decidability problems can be answered.

In this paper we will define certain subclasses of FDFD's, which we call Monogeneous RDFD's, Linear RDFD's, and Topologically Free Choice RDFD's. We will show that two of these three subclasses of FDFD's can be simulated via isomorphism by the correspondingly named subclasses of FIFO Petri Nets. It is known that isomorphisms between computation systems guarantee the same answers to corresponding decidability problems (e. g., reachability, deadlock, liveness) in the two systems ([KM82]). This means that problems where it is known that they can (not) be solved for a subclass of FIFO Petri Nets it follows immediately that the same problems can (not) be solved for the correspondingly named subclass of FDFD's.

Paper 4: Timed Data Flow Diagrams

Data Flow Diagrams (DFD's) are widely used in industry to express requirements specifications. However, as used in practice, there has been no precise semantics for DFD's, let alone an incorporation of a model of time. In this paper, we augment the Formalized Data Flow Diagrams (FDFD's) defined in [LWBL96] by adding a deterministic (or stochastic) time behavior for the consumption of values from in–flows to processes and the production of values to the out–flows from processes. We call our new FDFD model Timed (or Stochastic) Data Flow Diagrams (TDFD's or SDFD's). We identify two factors in determining how time can affect the choice of how an FDFD can change state. The first factor has to do with when the decision is made as to which state transition will be next occur. The two possibilities are a *Preselection Policy* and a *Race Policy*. The other timing factor is the past history of an FDFD execution. We identify three alternatives: *Resampling*, *Work Age Memory*, and *Enabling Age Memory*. Certain combinations of these alternatives allow us to model systems where components are competing for limited resources. Other combinations allow us to model systems where components work concurrently. Preemption can also be modeled using these alternatives. Major results for the quantitative and qualitative analysis of TDFD's can be borrowed from the literature on Timed Petri Nets.

Paper 5: Stochastic Analysis of Periodic Timed Data Flow Diagrams with Markovian Transition Times

Timed (or Stochastic) Data Flow Diagrams (TDFD's or SDFD's) introduced in [SB96d] are an extension of the Formalized Data Flow Diagrams, defined in [LWBL96]. This extension allows us to assess the quantitative behavior (e. g., performance, throughput, average load of a bubble, etc.) as well as the qualitative behavior (e. g., deadlock, reachability, termination, finiteness, liveness, etc.), eventually depending on different types of transition times, for the system modeled through the TDFD. In this paper, we consider Markovian transition times for the consumption of in–flow items and for the production of items on the out–flow. Moreover, we require the TDFD to be periodic and irreducible and it must have a finite reachability set. For these models, we have been able to apply an aggregation principle of [Sch84], extended for periodic Markov chains by [Woo93], to efficiently determine stationary probabilities, expected waiting times, and limiting process probabilities.

2 FORMALIZED DATA FLOW DIAGRAMS AND THEIR RELATION TO OTHER COMPUTATIONAL MODELS

A paper submitted to the Journal of Computer and System Sciences

Jürgen Symanzik[1] and Albert L. Baker[2]

Abstract

One approach to the formalization of Data Flow Diagrams (DFD's) is presented by Coleman ([Col91], [CB94]) and Leavens, et al., [LWBL96]. These Formalized Data Flow Diagrams (FDFD's) can be viewed as another model of computation. This paper contains an analysis of the computational power of these FDFD's. We first consider the issue whether certain features of FDFD's affect their computational power. A Reduced Data Flow Diagram (RDFD) is an FDFD with no stores, finite domains for flow values, and no facility for testing for empty flows, but it may contain persistent flows. An RDFD without persistent flows is called a persistent flow-free Reduced Data Flow Diagram (PFF-RDFD). We show that PFF-RDFD's are Turing equivalent. The other features of FDFD's only add to the expressive power of FDFD's ([SB96b]). Therefore, any FDFD can be expressed as an PFF-RDFD. Our proof that PFF-RDFD's are Turing equivalent procedes as follows. We first show that any RDFD can be simulated by a FIFO Petri Net. We then show that any Program Machine can be simulated by an PFF-RDFD. It is known that FIFO Petri Nets and Program Machines both are Turing equivalent.

Keywords

Computational Power, Turing Machine, FIFO Petri Net, Program Machine.

---

[1] Primary researcher and author.
[2] Professor, Department of Computer Science, Iowa State University.

2.1 Introduction

Traditional Data Flow Diagrams (DFD's) are the cornerstone of the software development methodology known as "Structured Analysis" (SA) ([DeM78], [WM85a]), and they are probably the most widely used specification technique in industry today ([BB93]). DFD's are popular because of their graphical representation and their hierarchical structure. Thus, they are ideally suited for users with non-technical backgrounds and are commonly used to depict the static structure of information flow in a system.

Traditional DFD's consist of a set of bubbles and a set of labeled flows. Bubbles represent either processes in a concurrent system or sequential procedures and are usually drawn as circles, ovals, or boxes. Flows represent data paths and are drawn as arrows connecting the bubbles. Flows coming into a bubble are called inflows and flows leaving a bubble are called outflows. Informally, a bubble reads the information on its inflows, and produces new information on its outflows. There are two types of flows. Persistent flows are like shared variables whose values are written by the source bubble and read by the destination bubble. Consumable flows are modelled as unbounded FIFO queues, with the source bubble enqueueing values on the queue and the destination bubble dequeueing values.

Even though traditional DFD's are popular, they lack formality and do not provide a rigorous definition of system functionality. Numerous attempts to formalize DFD's have appeared in the technical literature, e. g., in [DeM78], [WM85a], [WM85b], [Har87], [TP89], [You89], [Har92], and [Har96]. We focus on the Formalized Data Flow Diagrams (FDFD's) developed by Coleman, Wahls, Baker, and Leavens in [Col91], [CB94], [WBL93], and [LWBL96]. We answer the following questions about the computational power of FDFD's. Do FDFD's have the same computational power as FIFO Petri Nets, another specification technique for concurrent and distributed systems? If so, to which class of FIFO Petri Nets are FDFD's equivalent? Are FDFD's Turing equivalent?

After definitions of FDFD's, Reduced Data Flow Diagrams (RDFD's), persistent flow-free RDFD's (PFF-RDFD's), FIFO Petri Nets, and Program Machines are given in Section 2.2, we answer the above questions in Section 2.3. Indeed, we will show that PFF-RDFD's and Turing Machines are equivalent. Actually, we will not directly compare PFF-RDFD's and Turing Machines, but instead provide simulations of RDFD's by FIFO Petri Nets and simulations of Program Machines by PFF-RDFD's. A summary and a preview of future work concludes this paper in Section 2.4.

2.2 Computational Models

In this section we define FDFD's, RDFD's, and PFF-RDFD's, summarize the definitions of FIFO Petri Nets and Program Machines, and introduce the overall concept of a Computation System.

2.2.1 FDFD's and RDFD's

We will use the formal definitions from [LWBL96] to introduce Formalized Data Flow Diagrams (FDFD's). The cited paper contains a more detailed explanation of the underlying operational semantics of FDFD's and an extended example.

In all our definitions, we use the notation $X_\perp$ for the set $X \cup \{\perp\}$, where $\perp$ means no information.

Definition (2.2.1.1): A *Formalized Data Flow Diagram* (FDFD) is a quintuple

$$FDFD = (B, FLOWNAMES, TYPES, P, F),$$

where B is a set of *bubbles*, $FLOWNAMES$ is a set of *flows*, $TYPES$ is a set of *types*, P is the set $\{persistent, consumable\}$ and $F = B \times FLOWNAMES \times TYPES \times B \times P$. The following notational convention for members from these domains is used: $b \in B, fn \in FLOWNAMES, T \in TYPES, p \in P, f \in F$. ■

Definition (2.2.1.2): We define the following auxiliary functions for describing the structure of an FDFD:

Function : Type
$Source : F \rightarrow B$
$FlowName : F \rightarrow FLOWNAMES$
$TypeOf : F \rightarrow TYPES$
$Target : F \rightarrow B$
$Consumable : F \rightarrow Boolean$
$Inputs : B \rightarrow PowerSet(F)$
$Outputs : B \rightarrow PowerSet(F)$
$TypeMeaning : TYPES \rightarrow Set$

■

In the previous definition, we think of a type as describing a set of objects. The set of objects associated with a type T is given by $TypeMeaning(T)$. By Set, we mean the class of all recursive sets.

Definition (2.2.1.3): We define the following domains describing the configuration of an FDFD:

Notation for Members $\in$ Name	= description
$m \in MODES$	$= \{idle, working\}$
$bm \in BubbleMode$	$= B \rightarrow MODES$
$o \in OBJECTS$	$= \bigcup_{T \in TYPES} TypeMeaning(T)$
$s \in WhatRead$	$= (F \rightarrow OBJECTS_{\perp})$
$r \in Read$	$= B \rightarrow WhatRead$
$fs \in FlowState$	$= F \rightarrow OBJECTS^*$
$\gamma = (bm, r, fs) \in \Gamma$	$= BubbleMode \times Read \times FlowState$

We call $\gamma = (bm, r, fs) \in \Gamma$ a *state* (or a *configuration*) of an FDFD. ■

Definition (2.2.1.4): Sequences of objects, i. e., elements of $(OBJECTS^*)_{\perp}$, are treated as FIFO queues using the following constants and operations:

$$
\begin{aligned}
\langle\rangle &: OBJECTS^* \\
Enq &: (OBJECTS^*)_{\perp} \times OBJECTS \rightarrow (OBJECTS^*)_{\perp} \\
IsEmpty &: (OBJECTS^*)_{\perp} \rightarrow Boolean_{\perp} \\
Head &: (OBJECTS^*)_{\perp} \rightarrow OBJECTS_{\perp} \\
Rest &: (OBJECTS^*)_{\perp} \rightarrow (OBJECTS^*)_{\perp}
\end{aligned}
$$

These operations are defined to satisfy the following equations for all $q \in OBJECTS^*$ and $o \in OBJECTS$.

$$
\begin{aligned}
Enq(\perp, o) &= \perp \\
IsEmpty(\perp) &= \perp \\
IsEmpty(\langle\rangle) &= true \\
IsEmpty(Enq(q, o)) &= false \\
Head(\perp) &= \perp \\
Head(\langle\rangle) &= \perp \\
Head(Enq(q, o)) &= \textbf{if } IsEmpty(q) \textbf{ then } o \textbf{ else } Head(q) \textbf{ fi} \\
Rest(\perp) &= \perp \\
Rest(\langle\rangle) &= \perp \\
Rest(Enq(q, o)) &= \textbf{if } IsEmpty(q) \textbf{ then } \langle\rangle \textbf{ else } Enq(Rest(q), o) \textbf{ fi}
\end{aligned}
$$

■

The things that the bubbles in an FDFD can do are defined through three curried functions *Enabled*, *Consume*, and *Produce* defined as follows:

Definition (2.2.1.5): Enablement of a bubble can depend on both the presence of values on input flows as well as on the values on such flows:

$$Enabled : B \rightarrow (FlowState \rightarrow Boolean_{\perp})$$

A bubble that is enabled can change its mode from *idle* to *working*. It reads some of its inflows, and consumes some of these. Only consumable flows may be consumed and consumption means removing the head of the sequence associated with the flow:

$$Consume : B \rightarrow ((FlowState \times Read) \rightarrow PowerSet(FlowState \times Read)).$$

A bubble in *working* mode can produce some output in the transition from *working* to *idle*:

$$Produce : B \rightarrow ((FlowState \times Read) \rightarrow PowerSet(FlowState \times Read)).$$

The notation $[x \mapsto y]g$ is an update to a function, g, and it is defined by the following equation.

$$[x \mapsto y]g \stackrel{\text{def}}{=} \lambda z \,.\, \textbf{if } z = x \textbf{ then } y \textbf{ else } g(z) \textbf{ fi}$$

The curried function $In(f, b)$ represents the changes that b makes to the flow state and read function by reading the flow f, and it is defined as follows:

$In : (F \times B) \rightarrow ((FlowState \times Read) \rightarrow (FlowState \times Read))$

$In(f, b)(fs, r) =$

$\quad$ **let** $r_b = [f \mapsto Head(fs(f))](r(b))$

$\quad$ **in** (**if** $Consumable(f)$ **then** $[f \mapsto Rest(fs(f))]fs$ **else** fs **fi**,

$\quad\quad [b \mapsto r_b]r)$

By analogy, the curried function $Out(o, f, b)$ represents the changes that b makes to the flow state and read function by producing o on the flow f, and it is defined as follows:

$Out : (OBJECTS \times F \times B) \rightarrow ((FlowState \times Read) \rightarrow (FlowState \times Read))$

$Out(o, f, b)(fs, r) =$

$\quad$ **let** $r_b = \lambda f' \,.\, \perp$

$\quad$ **in** (**if** $Consumable(f)$ **then** $[f \mapsto Enq(fs(f), o)]fs$ **else** $[f \mapsto Enq(\langle\rangle, o)]fs$ **fi**,

$\quad\quad [b \mapsto r_b]r)$ ■

Two kinds of transitions are allowed between configurations: an enabled bubble can go from *idle* to *working*, and a *working* bubble can go to *idle*. We use the symbol $\longrightarrow$ to state the binary relation between configurations.

The transition rule

$$\frac{\begin{array}{c} bm(b) = idle, \\ Enabled(b)(fs) = true, \\ bm' = [b \mapsto working]bm, \\ (fs', r') \in Consume(b)(fs, r) \end{array}}{(bm, r, fs) \longrightarrow (bm', r', fs')}$$

states that if b is *idle* and enabled, then it may change its mode to *working* and consume some of its inputs. Finally, if the conditions above the horizontal line hold, then the transition below the line may take place.

The transition rule

$$\frac{\begin{array}{c} bm(b) = working, \\ bm' = [b \mapsto idle]bm, \\ (fs', r') \in Produce(b)(fs, r) \end{array}}{(bm, r, fs) \longrightarrow (bm', r', fs')}$$

states that if b is *working*, then it may change its mode to *idle* and produce some outputs.

We next define what we mean by a Reduced Data Flow Diagram (RDFD). We do not explicitely state that there are no stores in RDFD's since stores are considered as an addition to, but not as a part of the FDFD's defined in [LWBL96].

Definition (2.2.1.6): An FDFD that obeys to the following restrictions is called a *Reduced Data Flow Diagram* (RDFD):

(i) The set $OBJECTS$, that contains all types of objects that may appear on flows, is finite.

(ii) The mappings *Enabled*, *Consume*, and *Produce* contain no higher logical constructor than First Order Predicate Calculus (FOPC) assertions over the values on the inflows and outflows of the bubble. Each assertion must be computable ([WBL93]).

(iii) The mappings $Enabled(b)$ and $Consume(b)$ do not make use of the positive form of the operation *IsEmpty*. Instead, it only occurs with a negation, i. e., $\neg IsEmpty(fs(f))$, where $f \in Inputs(b)$.

(iv) The mappings $Enabled(b)$ and $Consume(b)$ only make use of the *Head* element of a flow $f \in Inputs(b)$. Constructions such as "**if** $Head(Rest(fs(f))) = x$ **then** ... **fi**" are not allowed.

(v) Every flow is initialized, i. e., $\forall\ f \in F\ :\ fs_{initial}(f) \in (TypeOf(f))^*$. Also, $bm_{initial} = \lambda b\ .\ idle$ and $r_{initial} = \lambda b\ .\ \lambda f\ .\ \perp$. ■

The finiteness of $OBJECTS = \bigcup_{T \in TYPES} TypeMeaning(T)$ claimed in (i) guarantees that each simple primitive type and each structured primitive type only has a fixed number of elements. E. g., we can enumerate $TypeMeaning(INTEGER) = \{MININT, \ldots, MAXINT\}$ and assuming that T satisfies this criterion, then

$$TypeMeaning(Sequence\ of\ T) = \{s \mid length(s) \leq c, \forall\, k \leq c\ :\ s[k] \in T\}$$

for some constant c.

By (ii), we can determine the result of each expression for every possible combination of inputs for this expression. Especially, we can reformulate every expression in the mappings *Enabled*, *Consume*, and *Produce* as finite many **if − then − fi**–statements that cover possible combinations of inputs. However, these statements may be nondeterministic.

Therefore, we can rewrite the mappings *Enabled*, *Consume*, and *Produce* of any RDFD in the following normal form. Later, we will use this normal form to simulate RDFD's by FIFO Petri Nets. If not otherwise stated, we use this normal form for all examples throughout this paper.

Definition (2.2.1.7): The *normal form* of an RDFD (*nf-RDFD*) is defined as follows:

$B_{FDFD} = \{b_1, \ldots, b_b\}$

$FLOWNAMES_{FDFD} = \{f_1, \ldots, f_f\}$

A bubble b_i can be never enabled (case E_0 below), it can be always enabled (case E_1), or it is enabled if at least one of its m_i enabling conditions of its enabling rule yields *true* (case E_2). An enabling condition j will yield *true* if the n_{ij1} consumable inflows $f^c_{ij1}, \ldots, f^c_{ijn_{ij1}}$ are not empty, the head elements of these inflows are the values $o^c_{ij1}, \ldots, o^c_{ijn_{ij1}}$, and the data on the n_{ij2} persistent inflows $f^p_{ij1}, \ldots, f^p_{ijn_{ij2}}$ are the values $o^p_{ij1}, \ldots, o^p_{ijn_{ij2}}$. An enabling condition may access only consumable flows (then $n_{ij2} = 0$), only persistent flows (then $n_{ij1} = 0$), or consumable and persistent flows (then $n_{ij1} > 0$ and $n_{ij2} > 0$). Also, f^c_{ijk} and f^c_{ilm} as well as f^p_{ijk} and f^p_{ilm} may be identical for some combination of (j, k) and (l, m) pairs.

$\forall\, b_i \in B\ :$

$$Enabled(b_i) = \lambda fs\ .\ false \qquad (E_0)$$

or

$$Enabled(b_i) = \lambda fs\ .\ true \qquad (E_1)$$

or

$$Enabled(b_i) = \lambda fs\ . \qquad (E_2)$$

$$(\neg IsEmpty(fs(f^c_{i11})) \wedge \ldots \wedge \neg IsEmpty(fs(f^c_{i1n_{i11}}))$$

$$\wedge Head(fs(f^c_{i11})) = o^c_{i11} \wedge \ldots \wedge Head(fs(f^c_{i1n_{i11}})) = o^c_{i1n_{i11}}$$

$$\wedge Head(fs(f^p_{i11})) = o^p_{i11} \wedge \ldots \wedge Head(fs(f^p_{i1n_{i12}})) = o^p_{i1n_{i12}})$$

$$\vee \ldots \vee$$

$$(\neg IsEmpty(fs(f^c_{im_i1})) \wedge \ldots \wedge \neg IsEmpty(fs(f^c_{im_in_{im_i1}}))$$

$$\wedge Head(fs(f^c_{im_i1})) = o^c_{im_i1} \wedge \ldots \wedge Head(fs(f^c_{im_in_{im_i1}})) = o^c_{im_in_{im_i1}}$$

$$\wedge Head(fs(f^p_{im_i1})) = o^p_{im_i1} \wedge \ldots \wedge Head(fs(f^p_{im_in_{im_i2}})) = o^p_{im_in_{im_i2}})$$

where $\forall\, i \in \{1, \ldots, b\}$:

$$m_i \geq 1,\ n_{i11}, n_{i12}, \ldots, n_{im_i1}, n_{im_i2} \geq 0$$

$$F^c_i = \{f^c_{i11}, \ldots, f^c_{i1n_{i11}}, \ldots, f^c_{im_i1}, \ldots, f^c_{im_in_{im_i1}}\}$$

$$F^p_i = \{f^p_{i11}, \ldots, f^p_{i1n_{i12}}, \ldots, f^p_{im_i1}, \ldots, f^p_{im_in_{im_i2}}\}$$

$$F_i = F^c_i \cup F^p_i = Inputs(b_i)$$

$$\forall\, f^c \in F^c_i\ :\ Consumable(f^c)$$

$$\forall\, f^p \in F^p_i\ :\ \neg Consumable(f^p)$$

$$O^c_i = \{o^c_{i11}, \ldots, o^c_{i1n_{i11}}, \ldots, o^c_{im_i1}, \ldots, o^c_{im_in_{im_i1}}\}$$

$$O^p_i = \{o^p_{i11}, \ldots, o^p_{i1n_{i12}}, \ldots, o^p_{im_i1}, \ldots, o^p_{im_in_{im_i2}}\}$$

$$O_i = O^c_i \cup O^p_i$$

$$\bigcup_{i=1,\ldots,b} F_i \subset F_{FDFD}$$

$$\bigcup_{i=1,\ldots,b} O_i \subset OBJECTS$$

A bubble b_i may consume nothing if it is never enabled or always enabled (case C_1 below). Otherwise, if several of the m_i enabling conditions of its enabling rule are *true*, b_i nondeterministically selects one of these *Consume* cases, say j, and it consumes from the n_{ij1} consumable inflows $f^c_{ij1}, \ldots, f^c_{ijn_{ij1}}$ and the n_{ij2} persistent inflows $f^p_{ij1}, \ldots, f^p_{ijn_{ij2}}$ of b_i referred to in this *Consume* case (case C_2). This implies that the head elements of the consumable inflows are removed from their queue. The persistent inflows are not modified at all.

$\forall\, b_i \in B$:

$$Consume(b_i) = \lambda(fs, r)\ .\ \{(fs, r)\} \qquad (C_1)$$

or

$$Consume(b_i) = \lambda(fs, r)\,. \qquad (C_2)$$

$$\{\textbf{if } \neg IsEmpty(fs(f^c_{i11})) \wedge \ldots \wedge \neg IsEmpty(fs(f^c_{i1n_{i11}}))$$

$$\wedge Head(fs(f^c_{i11})) = o^c_{i11} \wedge \ldots \wedge Head(fs(f^c_{i1n_{i11}})) = o^c_{i1n_{i11}}$$

$$\wedge Head(fs(f^p_{i11})) = o^p_{i11} \wedge \ldots \wedge Head(fs(f^p_{i1n_{i12}})) = o^p_{i1n_{i12}})$$

$$\textbf{then } In(f^c_{i11}, b_i)(\ldots(In(f^c_{i1n_{i11}}, b_i)(In(f^p_{i11}, b_i)(\ldots(In(f^p_{i1n_{i12}}, b_i)(fs, r))\ldots)))\ldots)$$

$$\textbf{fi},$$

$$\ldots,$$

$$\textbf{if } \neg IsEmpty(fs(f^c_{im_i1})) \wedge \ldots \wedge \neg IsEmpty(fs(f^c_{im_in_{im_i1}}))$$

$$\wedge Head(fs(f^c_{im_i1})) = o^c_{im_i1} \wedge \ldots \wedge Head(fs(f^c_{im_in_{im_i1}})) = o^c_{im_in_{im_i1}}$$

$$\wedge Head(fs(f^p_{im_i1})) = o^p_{im_i1} \wedge \ldots \wedge Head(fs(f^p_{im_in_{im_i2}})) = o^p_{im_in_{im_i2}})$$

$$\textbf{then } In(f^c_{im_i1}, b_i)(\ldots(In(f^c_{im_in_{im_i1}}, b_i)(In(f^p_{im_i1}, b_i)(\ldots(In(f^p_{im_in_{im_i2}}, b_i)(fs, r))\ldots)))\ldots)$$

$$\textbf{fi}$$

$$\}$$

A bubble b_i may produce nothing and simply reset its internal status independently from what it has read (case P_0 below). It may nondeterministically select one of its s_i *Produce* cases, say j, producing objects $u_{ij1}, \ldots, u_{ijk_{ij}}$ on its outflows $h_{ij1}, \ldots, h_{ijk_{ij}}$ (case P_1). Otherwise, if it has read some input that matches one of the m_i *Consume* cases, say j, it nondeterministically selects one of the l_{ij} *Produce* subcases, say k, and produces objects $u_{ijk1}, \ldots, u_{ijkq_{ijk}}$ on its outflows $h_{ijk1}, \ldots, h_{ijkq_{ijk}}$ (case P_2). We use the symbol $\square$ to indicate that any of the l_{ij} *Produce* subcases can be selected nondeterministically if *Produce* case j has been selected (nondeterministically).

$\forall\, b_i \in B$:

$$Produce(b_i) = \lambda(fs, r)\,.\,\{(fs, [b_i \mapsto \lambda f\,.\,\perp]r)\} \qquad (P_0)$$

or

$$Produce(b_i) = \lambda(fs, r)\,. \qquad (P_1)$$

$$\{Out(u_{i11}, h_{i11}, b_i)(\ldots(Out(u_{i1k_{i1}}, h_{i1k_{i1}}, b_i)(fs, r))\ldots),$$

$$\ldots,$$

$$Out(u_{is_i1}, h_{is_i1}, b_i)(\ldots(Out(u_{is_ik_{is_i}}, h_{is_ik_{is_i}}, b_i)(fs, r))\ldots)\}$$

$$\}$$

or

$$Produce(b_i) = \lambda(fs, r)\,. \qquad (P_2)$$

$$\{\textbf{if } r(b_i)(f^c_{i11}) = o^c_{i11} \wedge \ldots \wedge r(b_i)(f^c_{i1n_{i11}}) = o^c_{i1n_{i11}}$$

$$\wedge r(b_i)(f^p_{i11}) = o^p_{i11} \wedge \ldots \wedge r(b_i)(f^p_{i1n_{i12}}) = o^p_{i1n_{i12}}$$

then $Out(u_{i111}, h_{i111}, b_i)(\ldots(Out(u_{i11q_{i11}}, h_{i11q_{i11}}, b_i)(fs, r))\ldots)$

$\square \ldots$

$\vdots$

$\square Out(u_{i1l_{i1}1}, h_{i1l_{i1}1}, b_i)(\ldots(Out(u_{i1l_{i1}q_{i1l_{i1}}}, h_{i1l_{i1}q_{i1l_{i1}}}, b_i)(fs, r))\ldots)$

fi,

$\ldots$,

if $r(b_i)(f^c_{im_i1}) = o^c_{im_i1} \wedge \ldots \wedge r(b_i)(f^c_{im_in_{im_i}}) = o^c_{im_in_{im_i}}$

$\wedge r(b_i)(f^p_{im_i1}) = o^p_{im_i1} \wedge \ldots \wedge r(b_i)(f^p_{im_in_{im_i2}}) = o^p_{im_in_{im_i2}}$

then $Out(u_{im_i11}, h_{im_i11}, b_i)(\ldots(Out(u_{im_i1q_{im_i1}}, h_{im_i1q_{im_i1}}, b_i)(fs, r))\ldots)$

$\square \ldots$

$\vdots$

$\square Out(u_{im_il_{im_i}1}, h_{im_il_{im_i}1}, b_i)(\ldots(Out(u_{im_il_{im_i}q_{im_il_{im_i}}}, h_{im_il_{im_i}q_{im_il_{im_i}}}, b_i)(fs, r))\ldots)$

fi

}

where $\forall i \in \{1, \ldots, b\}$:

$m_i \geq 1,\ l_{i1}, \ldots, l_{im_i}, q_{i11}, \ldots, q_{im_il_{im_i}} \geq 0$

$H_i = \{h_{i111}, \ldots, h_{i11q_{i11}}, \ldots, h_{im_il_{im_i}1}, \ldots, h_{im_il_{im_i}q_{im_il_{im_i}}}\}$

$U_i = \{u_{i111}, \ldots, u_{i11q_{i11}}, \ldots, u_{im_il_{im_i}1}, \ldots, u_{im_il_{im_i}q_{im_il_{im_i}}}\}$

$\bigcup_{i=1,\ldots,b} H_i \subset F_{FDFD}$

$\bigcup_{i=1,\ldots,b} U_i \subset OBJECTS$

■

We assume $Consume(b_i)$ and $Produce(b_i)$ are finite enumerable and ordered sets for every $b_i \in B$. Thus, we can access the jth element of $Consume(b_i)$ through $(Consume(b_i))_j$ and the kth element of $Produce(b_i)$ through $(Produce(b_i))_k$.

Definition (2.2.1.8): An RDFD that does not have any persistent flows is called a *persistent flow-free Reduced Data Flow Diagram* (PFF-RDFD). This implies that $F = B \times FLOWNAMES \times TYPES \times B \times \{consumable\}$. ■

Definition (2.2.1.9): A *firing sequence* (*computation sequence*) of an FDFD is a possibly infinite sequence $(b_i, a_i, j_i) \in B \times \{C, P\} \times \mathbb{N}, i \geq 0$, such that, if transition (b_i, a_i, j_i) is fired in state (bm, r, fs), then

$$(fs', r') = \begin{cases} (Consume(b_i))_{j_i}(fs, r), & \text{if } a_i = C \\ (Produce(b_i))_{j_i}(fs, r), & \text{if } a_i = P \end{cases}$$

$$bm'(b_i) = \begin{cases} working, & \text{if } a_i = C \\ idle, & \text{if } a_i = P \end{cases}$$

$$bm'(b) = bm(b) \quad \forall b \in B - \{b_i\}$$

and

$$(bm, r, fs) \rightarrow (bm', r', fs').$$

We introduce the notation $(bm, r, fs)[(b, a, j)]$ to indicate that transition (b, a, j) is fireable in state (bm, r, fs) and $(bm, r, fs)[(b, a, j)](bm', r', fs')$ to indicate that state (bm', r', fs') is reached upon the firing of transition (b, a, j) in state (bm, r, fs).

By induction, we extend this notation for firing sequences:

$$(bm_0, r_0, fs_0)[(b_1, a_1, j_1), \ldots, (b_{n-1}, a_{n-1}, j_{n-1}), (b_n, a_n, j_n)]$$

is used to indicate that transition (b_n, a_n, j_n) is fireable in state $(bm_{n-1}, r_{n-1}, fs_{n-1})$, given that

$$(bm_0, r_0, fs_0)[(b_1, a_1, j_1), \ldots, (b_{n-1}, a_{n-1}, j_{n-1})](bm_{n-1}, r_{n-1}, fs_{n-1})$$

holds. By analogy, we use

$$(bm_0, r_0, fs_0)[(b_1, a_1, j_1), \ldots, (b_n, a_n, j_n)](bm_n, r_n, fs_n)$$

to indicate that state (bm_n, r_n, fs_n) is reached upon the firing of the sequence $(b_1, a_1, j_1), \ldots, (b_n, a_n, j_n)$.

■

Definition (2.2.1.10): The *set of firing sequences* (*set of computation sequences, language*) of an FDFD, denoted by $FS(FDFD, \gamma_{initial})$, is the set containing all firing sequences that are possible for this FDFD, given $\gamma_{initial} = (bm_{initial}, r_{initial}, fs_{initial}) = (bm_0, r_0, fs_0)$, i. e.,

$$FS(FDFD, \gamma_{initial}) = \{s \mid s \in (B \times \{C, P\} \times \mathbb{N})^* \wedge \gamma_{initial}[s]\}.$$ ■

Definition (2.2.1.11): The *Reachability Set* of an FDFD, denoted by $RS(FDFD, \gamma_{initial})$, is the set of states $\gamma = (bm, r, fs)$ that are reachable from $\gamma_{initial} = (bm_{initial}, r_{initial}, fs_{initial})$, i. e.,

$$RS(FDFD, \gamma_{initial}) = \{\gamma \mid \gamma \in \Gamma \wedge \exists s \in FS(FDFD, \gamma_{initial}) : \gamma_{initial}[s]\gamma\}.$$ ■

2.2.2 FIFO Petri Nets

In some sense, FIFO Petri Nets (introduced in [MM81]) are Petri Nets where places contain words instead of tokens and arcs are labelled by words. More formally, we make use of the definition of FIFO Petri Nets as given in [Rou87].

Definition (2.2.2.1): A *FIFO Petri Net* is a quintuple $FPN = (P, T, B, F, Q)$ where P is a finite set of *places* (also called *queues*), T is a finite set of *transitions* (disjoint from P), Q is a finite *queue alphabet*, and $F : T \times P \to Q^*$ and $B : P \times T \to Q^*$ are two mappings called respectively *forward* and *backward incidence mappings.* ■

Definition (2.2.2.2): A *marking* M of a FIFO Petri Net is a mapping $M : P \to Q^*$.

A transition t is *fireable* in M, written $M(t>$, if $\forall p \in P : B(p,t) \leq M(p)$ (where $u \leq x$ means u is a prefix of x).

For a marking M, we define the firing of a transition t, written $M(t > M'$, if $M(t >$ and the following equation between words holds $\forall p \in P : B(p,t)M'(p) = M(p)F(t,p)$. That means, the firing of a transition t removes $B(p,t)$ from the head of $M(p)$ and appends $F(t,p)$ to the end of the resulting word. ■

Definition (2.2.2.3): A FIFO Petri Net FPN together with an initial marking $M_0 : P \to Q^*$ is called a *marked FIFO Petri Net* and is denoted by (FPN, M_0).

As usual, the firing of a transition can be extended to the firing of a sequence of transitions. We denote by $FS(FPN, M_0)$ the *set of firing sequences* of this FIFO Petri Net. The firing of a sequence u of transitions from a marking M to a marking M' is written as $M(u > M'$.

The set of markings that are reachable from M_0 is called *Reachability Set* and it is denoted by $Acc(FPN, M_0)$. ■

2.2.3 Program Machines

We introduce Program Machines as given in [Min67], using the formalism of [VVN81].

Definition (2.2.3.1): A *Program Machine* is given by a finite set $R = \{r_1, \ldots, r_p\}$ of *registers*, a finite set $Q = \{q_0, \ldots, q_r\}$ of *labels*, and a finite set I of *instructions*. Each instruction is labelled by an element of Q and no two instructions have the same label. A Program Machine has exactly one "start

instruction"

$$q_0 : \textbf{start goto } q_1;$$

and exactly one "halt instruction"

$$q_f : \textbf{halt.}$$

The other instructions are of the type "increment register r_i"

$$q_s : r_i := r_i + 1 \textbf{ goto } q_m;$$

or "test and decrement register r_i"

$$q_s : \textbf{if } r_i = 0 \textbf{ then goto } q_l \textbf{ else } r_i := r_i - 1 \textbf{ goto } q_m.$$

The registers have nonnegative integers as values. First the start instruction is executed. Then the flow of control is determined by the goto contained in each instruction, until the halt instruction is reached.

■

Definition (2.2.3.2): A *computation sequence* of a Program Machine is a possibly infinite sequence

$$q_0, (k_1^1, \ldots, k_1^p, q_1), (k_2^1, \ldots, k_2^p, q_{i_2}), \ldots, (k_j^1, \ldots, k_j^p, q_{i_j}), \ldots, (k_r^1, \ldots, k_r^p, q_{i_r})$$

with $q_{i_r} = q_f$, i. e., the halt statement is reached. q_{i_j} is the actual instruction to be executed in step j and k_j^s is the content of register r_s just before the execution of instruction q_{i_j} in step j. In computations, arbitrary initial values $(k_1^1, \ldots, k_1^p) \in \mathbb{N}^p$ are allowed. Note that computations may be of infinite length.

■

2.2.4 Computation Systems and Homomorphisms

Now we will introduce the overall concept of a Computation System. All the definitions and results in this subsection are drawn from [KM82]. However, there exist similar approaches in the literature, e. g., in [Jen80], where the terms "transition system", "simulation", and "consistent homomorphism" have been defined with respect to a formal method that allows comparisons of the descriptive power of different types of Petri Nets.

For any set Σ, we denote by Σ^* the set of finite sequences of elements of Σ including the null sequence λ. The set of infinite sequences of elements of Σ is denoted by Σ^∞ and we define $\Sigma^\omega = \Sigma^\infty \cup \Sigma^*$.

Definition (2.2.4.1): A *computation system* $S = (\Sigma, D, x, \overline{\ })$ consists of a set D, an element x of D, a finite set Σ of *operations*, and a function "$\overline{\ }$" from Σ to the set of partial functions from D to

D. That is, for each $a \in \Sigma$, $\overline{a}$ is a partial function from D to D. The function "$\overline{\ }$" is extended to Σ^* by $\overline{\lambda} =$ identity, $\overline{\alpha\beta}(y) = \overline{\alpha} \cdot \overline{\beta}(y) = \overline{\beta}(\overline{\alpha}(y))$, $\alpha \cdot \beta \in \Sigma^*, y \in D$. ■

Here D is intuitively thought of as the set of states (or configurations) of the computation system, where a state includes control information as well as data for synchronization. The element x is then considered to be the initial state of the system. The performance of an operation will create a new state, as defined by the function $\overline{\ }$, and a sequence of operation performances can be thought of as a computation (or firing) sequence of the system.

Definition (2.2.4.2): Let $S = (\Sigma, D, x, \overline{\ })$ be a computation system. Let $y, z \in D$, and $\alpha \in \Sigma^*$. We define:

$$\overline{\alpha}(y) = \perp, \quad \text{if } \overline{\alpha}(y) \text{ is undefined}$$
$$\overline{\alpha}(y) \neq \perp, \quad \text{if } \overline{\alpha}(y) \text{ is defined}$$
$$y \xrightarrow{\alpha} z, \quad \text{if } \overline{\alpha}(y) = z$$
$$y \xrightarrow{*} z, \quad \text{if } \exists \alpha \in \Sigma^* : y \xrightarrow{\alpha} z$$ ■

Definition (2.2.4.3): Let $S = (\Sigma, D, x, \overline{\ })$ be a computation system. For each $y \in D$, a *computation sequence* from y is a (finite or infinite) sequence $a_1 a_2 \ldots$ of elements of Σ such that $\forall i : \overline{a_1 a_2 \ldots a_i}(y) \neq \perp$. We denote by $C_S^\omega(y)$ the *set of all computation sequences from* y and by $C_S(y) = \{\alpha \mid \alpha \in \Sigma^*, \overline{\alpha}(y) \neq \perp\}$ the *set of all finite computation sequences from* y. For each $y \in D$, we denote by $R_S(y) = \{\overline{\alpha}(y) \mid \alpha \in \Sigma^*, \overline{\alpha}(y) \neq \perp\}$ the *reachability set* from y. When $y = x$, we simply write R_S, C_S, and C_S^ω instead of $R_S(x)$, $C_S(x)$, and $C_S^\omega(x)$, respectively. ■

Note that for an $\alpha \in \Sigma^\omega$, α is in $C_S^\omega(y)$ if, and only if, every prefix of α is in $C_S(y)$.

The following definitions give some ideas about possible relations between different computation systems.

Definition (2.2.4.4): Let $S_1 = (\Sigma_1, D_1, x_1, \overline{\ }^1)$ and $S_2 = (\Sigma_2, D_2, x_2, \overline{\ }^2)$ be computation systems. A *homomorphism* $h = (\tau, \rho) : S_1 \to S_2$ consists of a homomorphism $\tau : \Sigma_1^* \to \Sigma_2^*$, and an injection $\rho : D_1 \to D_2$ which satisfies the following conditions:

$$\rho(x_1) = x_2 \qquad (2.2.4.4.1)$$

$$\forall y, z \in R_{S_1}, \forall \alpha \in \Sigma_1^* : (y \xrightarrow{\alpha} z \Rightarrow \rho(y) \xrightarrow{\tau(\alpha)} \rho(z)) \qquad (2.2.4.4.2)$$

- A homomorphism is said to be *injective* if $\tau : \Sigma_1^* \rightarrow \Sigma_2^*$ is injective.
- A homomorphism is said to be *surjective* if the following conditions hold:

$$\forall y, z \in R_{S_1} \forall \beta \in \Sigma_2^* : (\rho(y) \xrightarrow{\beta} \rho(z) \Rightarrow \exists \alpha \in \Sigma_1^* : (\beta = \tau(\alpha) \wedge y \xrightarrow{\alpha} z)) \qquad (2.2.4.4.3)$$

$$\forall y \in R_{S_1} \forall z' \in D_2 : (\rho(y) \xrightarrow{*} z' \Rightarrow \exists z \in R_{S_1} : z' \xrightarrow{*} \rho(z)) \qquad (2.2.4.4.4)$$

- A homomorphism is said to be *bijective* if it is surjective and injective. ■

The injection ρ relates elements of D_1 to elements of D_2, where condition (2.2.4.4.1) states that the starting state of S_1 maps into the starting state of S_2. By condition (2.2.4.4.2), if α is any computation sequence in S_1 then there is a related computation sequence $\tau(\alpha)$ in S_2, with appropriate state mappings using ρ. Condition (2.2.4.4.3) means that any computation from $\rho(y)$ to $\rho(z)$ in S_2 is the image of a computation from y to z in S_1. Condition (2.2.4.4.4) means that any computation from $\rho(y)$ in S_2 is an initial segment of the image of some computation from y in S_1.

Definition (2.2.4.5): Let $S = (\Sigma, D, x, \overline{\ })$ be a computation system. Let $\alpha, \beta \in \Sigma^*$. We write $\alpha \simeq \beta$ if α is a permutation of β. Let $h = (\tau, \rho) : S_1 \rightarrow S_2$ be a homomorphism. h is said to be *spanning* if the following conditions hold:

$$\forall y, z \in R_{S_1} \forall \beta \in \Sigma_2^* : (\rho(y) \xrightarrow{\beta} \rho(z) \Rightarrow \exists \alpha \in \Sigma_1^* : \beta \simeq \tau(\alpha) \wedge y \xrightarrow{\alpha} z) \qquad (2.2.4.5.1)$$

$$\exists k \in I\!N \forall y \in R_{S_1} \forall y' \in D_2 \forall \beta \in \Sigma_2^* : (\rho(y) \xrightarrow{\beta} y' \Rightarrow \exists \alpha \in \Sigma_1^* \exists z \in D_1 \exists z' \in D_2 \exists \beta', \beta'' \in \Sigma_2^* :$$

$$\rho(y) \xrightarrow{\tau(\alpha)} \rho(z) \xrightarrow{\beta''} z' \wedge y' \xrightarrow{\beta'} z' \wedge \tau(\alpha)\beta'' \simeq \beta\beta' \wedge \mid \beta'' \mid \leq k) \qquad (2.2.4.5.2)$$

$$(z \xrightarrow{\gamma} u \text{ in } S_1) \Rightarrow \exists u' \in D_2 \exists \gamma', \gamma'' \in \Sigma_2^* : z' \xrightarrow{\gamma'} u' \wedge \rho(u) \xrightarrow{\gamma''} u' \wedge \tau(\gamma)\gamma'' \simeq \beta''\gamma' \qquad (2.2.4.5.3)$$

■

Condition (2.2.4.5.2) means that any computation β from $\rho(y)$ of S_2 is a prefix of a permutation of a computation of the form $\tau(\alpha)\beta'', \alpha \in \Sigma_1^*, \mid \beta'' \mid \leq k$. If $\alpha\gamma$ is a computation from y of S_1, then there must be a computation of the form $\beta\beta'\gamma'$ such that $\tau(\alpha)\tau(\gamma)$ is a prefix of a permutation of $\beta\beta'\gamma'$. Intuitively, β is an initial segment of a simulation of $\tau(\alpha)$, and if $\alpha\gamma$ is a computation of S_1, then β can be followed by a computation to simulate $\tau(\gamma)$.

Theorem (2.2.4.6): Every surjective homomorphism is a spanning homomorphism.

Definition (2.2.4.7): Let $h = (\tau, \rho) : S_1 \to S_2$ be a homomorphism. h is said to be *length preserving* if $\forall a \in \Sigma_1 : \mid \tau(a) \mid = 1$. ■

Definition (2.2.4.8): Let Σ be a finite set. For each $w \in \Sigma^*$, let $\iota(w)$ be the subset of Σ defined by $\iota(w) = \{a \mid \alpha a \beta = w, \alpha, \beta \in \Sigma^*, a \in \Sigma\}$.

Let $h = (\tau, \rho) : S_1 \to S_2$ be a surjective homomorphism. h is said to be *principal* if for each a and b in Σ_1, either $\tau(a) = \tau(b)$ or $\iota(\tau(a)) \cap \iota(\tau(b)) = \emptyset$, and $\overline{a} \neq \overline{b}$ implies $\iota(\tau(a)) \cap \iota(\tau(b)) = \emptyset$. ■

Theorem (2.2.4.9): Let $h = (\tau, \rho) : S_1 \to S_2$ be a bijective homomorphism. For each $y, z \in R_{S_1}$ and $\alpha \in \Sigma_1^*$ it holds that

$$y \xrightarrow{\alpha} z \text{ if, and only if, } \rho(y) \xrightarrow{\tau(\alpha)} \rho(z). \qquad (2.2.4.9.1)$$

Definition (2.2.4.10): Let $h = (\tau, \rho) : S_1 \to S_2$ be a homomorphism. h is called an *isomorphism* if there is a homomorphism $h' = (\tau', \rho') : S_2 \to S_1$ such that $hh' : S_2 \to S_2$ and $h'h : S_1 \to S_1$ are identities, i. e., $\tau\tau' : \Sigma_2^* \to \Sigma_2^*$, $\tau'\tau : \Sigma_1^* \to \Sigma_1^*$, $\rho\rho' : D_2 \to D_2$, and $\rho'\rho : D_1 \to D_1$ are identities. ■

Theorem (2.2.4.11): Let $S_1 = (\Sigma_1, D_1, x_1, \overline{}^1)$ and $S_1 = (\Sigma_2, D_2, x_2, \overline{}^2)$ be computation systems. Let $\tau : \Sigma_1^* \to \Sigma_2^*$ be a homomorphism and $\rho : D_1 \to D_2$ be a function. Then $h = (\tau, \rho)$ is an isomorphism from S_1 to S_2 if, and only if, τ and ρ are bijective and satisfy conditions (2.2.4.4.1) and (2.2.4.9.1).

Definition (2.2.4.12): A computation system $S = (\Sigma, D, x, \overline{})$ is said to be *reduced* if $D = R_S$. For each computation system $S = (\Sigma, D, x, \overline{})$, the computation system $\hat{S} = (\Sigma, R_S, x, \overline{})$ is called the *reduced form of* S, where for each $a \in \Sigma$, the partial function $\overline{a}$ in $\hat{S}$ is the restriction $\overline{a} \mid R_S$ of $\overline{a} \in S$. ■

Theorem (2.2.4.13): Let S_1 and S_2 be reduced computation systems. Let $h = (\tau, \rho) : S_1 \to S_2$ be a homomorphism. h is an isomorphism if, and only if, h is length preserving and bijective.

2.3 Equivalence of PFF–RDFD's and Turing Machines

In this section, we will show that PFF–RDFD's and Turing Machines are equivalent with respect to their computational power. Since we know that Turing Machines, Program Machines, and FIFO Petri

Nets are equivalent ([Min67], [FM82], and [MF85], respectively), it is sufficient to show that we can simulate each RDFD, and hence each PFF–RDFD, by a FIFO Petri Net and each Program Machine by a PFF–RDFD. Actually, the first simulation could be omitted and instead, we could simply invoke Church's Thesis to deduce that PFF–RDFD's have at most the computational power of Turing Machines (and thus of FIFO Petri Nets). However, we will show a little bit more than only the existence of a simulation, but also that the simulation of RDFD's (and thus of PFF–RDFD's) by FIFO Petri Nets is based on an isomorphism. For the other direction, a PFF–RDFD is sufficient to simulate a Program Machine.

Now, we will formally state our main result of this paper. The proof follows by the above and Theorems (2.3.1.1) and (2.3.2.1).

Theorem (2.3.1): PFF–RDFD's and Turing Machines have the same computational power.

2.3.1 Simulation of RDFD's by FIFO Petri Nets

Theorem (2.3.1.1): Every RDFD can be simulated by a FIFO Petri Net with respect to an isomorphism h.

Proof: Without loss of generality, we assume that our RDFD is given as a nf–RDFD (B_{RDFD}, $FLOWNAMES_{RDFD}$, $TYPES_{RDFD}$, P_{RDFD}, F_{RDFD}) with mappings *Enabled*, *Consume*, and *Produce* and with initial values $(bm_{initial}, r_{initial}, fs_{initial})$. We consider the computation system

$$S_{RDFD} = ((B_{RDFD}, \{C, P\}, I\!N), (bm, r, fs), (bm_{initial}, r_{initial}, fs_{initial}), \overset{RDFD}{\text{---}}).$$

We now construct a marked FIFO Petri Net $FPN = ((P_{FPN}, T_{FPN}, B_{FPN}, F_{FPN}, Q_{FPN}), M_{0,FPN})$ as follows:

For each flow in the RDFD we generate a place in the FIFO Petri Net. For an easier reference we assume that $\forall i = 1, \ldots, f, f_i \in FLOWNAMES_{RDFD} : FlowName(f_i) = f_i$. For each bubble in the RDFD we require additional places in the FIFO Petri Net to store the bubble's working mode and the values that have been read. Whenever we refer to type C_1, C_2, P_0, P_1, and P_2, we mean that the *Consume* or *Produce* case is in the related normal form, introduced in Definition (2.2.1.7).

$$P_{FPN} = \{f_1, \ldots, f_f\}$$
$$\cup \{b_{1,idle}, \ldots, b_{b,idle}\}$$
$$\cup \bigcup_{i \in \{1,\ldots,b\}} \Big(\{b_{i,working\,1} \mid Consume(b_i) \text{ is of type } C_1\}$$

$$\cup\{b_{i,working:1}, \ldots, b_{i,working:m_i} \mid Consume(b_i) \text{ is of type } C_2,$$
$$m_i = (\#\text{ of cases in } Consume(b_i))\}\Big)$$

For each case (subcase) in the mappings *Consume* and *Produce*, we generate a transition in the FIFO Petri Net.

$$T_{FPN} = \bigcup_{i\in\{1,\ldots,b\}} \Big(\{c_{i1} \mid Consume(b_i) \text{ is of type } C_1\}$$
$$\cup\{c_{i1}, \ldots, c_{im_i} \mid Consume(b_i) \text{ is of type } C_2,\ m_i = (\#\text{ of cases in } Consume(b_i))\}$$
$$\cup\{p_{i1}, \ldots, p_{is_i} \mid Produce(b_i) \text{ is of type } P_1,\ s_i = (\#\text{ of cases in } Produce(b_i))\}$$
$$\cup\{p_{i11}, \ldots, p_{i1l_{i1}}, \ldots, p_{im_i1}, \ldots, p_{im_il_{im_i}} \mid Produce(b_i) \text{ is of type } P_2,$$
$$m_i = (\#\text{ of cases in } Produce(b_i)),\ l_{ij} = (\#\text{ of subcases in case } j \text{ in } Produce(b_i))\}\Big)$$

For each $b_i \in B_{RDFD}$, define the mappings B_{FPN} and F_{FPN} in accordance with the mappings *Consume* and *Produce*.

If $Consume(b_i)$ is of type C_1, then upon firing of transition c_{i1} the idle token I is removed from place $b_{i,idle}$ and the working token W is queued at place $b_{i,working:1}$. We define:

$$B(b_{i,idle}, c_{i1}) = I$$

$$F(c_{i1}, b_{i,working:1}) = W$$

If $Consume(b_i)$ is of type C_2, then upon firing of transition c_{ij} the idle token I is removed from place $b_{i,idle}$, the tokens $o^c_{ij1}, \ldots, o^c_{ijn_{ij1}}$ are removed from the places $f^c_{ij1}, \ldots, f^c_{ijn_{ij1}}$ representing consumable flows, the tokens $o^p_{ij1}, \ldots, o^p_{ijn_{ij2}}$ are removed from the places $f^p_{ij1}, \ldots, f^p_{ijn_{ij2}}$ representing persistent flows, the working token W is queued at place $b_{i,working:j}$ and the tokens $o^p_{ij1}, \ldots, o^p_{ijn_{ij2}}$ are queued at places $f^p_{ij1}, \ldots, f^p_{ijn_{ij2}}$ representing persistent flows. We define:

$$B(b_{i,idle}, c_{i1}) = I$$
$$\vdots$$
$$B(b_{i,idle}, c_{im_i}) = I$$

$$B(f^c_{i11}, c_{i1}) = o^c_{i11}$$
$$\vdots$$
$$B(f^c_{i1n_{i11}}, c_{i1}) = o^c_{i1n_{i11}}$$

$$
\begin{aligned}
B(f^p_{i11}, c_{i1}) &= o^p_{i11} \\
&\vdots \\
B(f^p_{i1n_{i12}}, c_{i1}) &= o^p_{i1n_{i12}} \\
&\vdots \\
B(f^c_{im_i1}, c_{im_i}) &= o^c_{im_i1} \\
&\vdots \\
B(f^c_{im_in_{im_i1}}, c_{im_i}) &= o^c_{im_in_{im_i1}} \\
B(f^p_{im_i1}, c_{im_i}) &= o^p_{im_i1} \\
&\vdots \\
B(f^p_{im_in_{im_i2}}, c_{im_i}) &= o^p_{im_in_{im_i2}}
\end{aligned}
$$

$$
\begin{aligned}
F(c_{i1}, b_{i,working:1}) &= W \\
&\vdots \\
F(c_{im_i}, b_{i,working:m_i}) &= W
\end{aligned}
$$

$$
\begin{aligned}
F(c_{i1}, f^p_{i11}) &= o^p_{i11} \\
&\vdots \\
F(c_{i1}, f^p_{i1n_{i12}}) &= o^p_{i1n_{i12}} \\
&\vdots \\
F(c_{im_i}, f^p_{im_i1}) &= o^p_{im_i1} \\
&\vdots \\
F(c_{im_i}, f^p_{im_in_{im_i2}}) &= o^p_{im_in_{im_i2}}
\end{aligned}
$$

If $Produce(b_i)$ is of type P_1, then upon firing of transition p_{ij} the working token W is removed from place $b_{i,working:j}$, the idle token I is queued at place $b_{i,idle}$, and the tokens $u_{ij1}, \ldots, u_{ijk_{ij}}$ are queued at places $h_{ij1}, \ldots, h_{ijk_{ij}}$. We define:

$$
\begin{aligned}
B(b_{i,working:1}, p_{i1}) &= W \\
&\vdots \\
B(b_{i,working:1}, p_{is_i}) &= W
\end{aligned}
$$

$$
\begin{aligned}
F(p_{i1}, b_{i,idle}) &= I \\
&\vdots \\
F(p_{is_i}, b_{i,idle}) &= I
\end{aligned}
$$

$$
\begin{aligned}
F(p_{i1}, h_{i11}) &= u_{i11} \\
&\vdots \\
F(p_{i1}, h_{i1k_{i1}}) &= u_{i1k_{i1}} \\
&\vdots \\
F(p_{is_i}, h_{is_i1}) &= u_{is_i1} \\
&\vdots \\
F(p_{is_i}, h_{is_ik_{is_i}}) &= u_{is_ik_{is_i}}
\end{aligned}
$$

If $Produce(b_i)$ is of type P_2, then upon firing of transition p_{ijk} the working token W is removed from place $b_{i,working:j}$, the idle token I is queued at place $b_{i,idle}$, and the tokens $u_{ijk1}, \ldots, u_{ijkq_{ijk}}$ are queued at places $h_{ijk1}, \ldots, h_{ijkq_{ijk}}$. Also, if place h_{ijk} represents a persistent flow, any value that is currently queued at this place will be removed upon firing of this transition. We define:

$$
\begin{aligned}
B(b_{i,working:1}, p_{i11}) &= W \\
&\vdots \\
B(b_{i,working:1}, p_{i1l_{i1}}) &= W \\
&\vdots \\
B(b_{i,working:m_i}, p_{im_i1}) &= W \\
&\vdots \\
B(b_{i,working:m_i}, p_{im_il_{im_i}}) &= W
\end{aligned}
$$

$$
\begin{aligned}
F(p_{i11}, b_{i,idle}) &= I \\
&\vdots \\
F(p_{i1l_{i1}}, b_{i,idle}) &= I \\
&\vdots \\
F(p_{im_i1}, b_{i,idle}) &= I
\end{aligned}
$$

$$\vdots$$

$$F(p_{im_i l_{im_i}}, b_{i,idle}) = I$$

$$F(p_{i11}, h_{i111}) = u_{i111}$$

$$\vdots$$

$$F(p_{i11}, h_{i11q_{i11}}) = u_{i11q_{i11}}$$

$$\vdots$$

$$F(p_{i1l_{i1}}, h_{i1l_{i1}1}) = u_{i1l_{i1}1}$$

$$\vdots$$

$$F(p_{i1l_{i1}}, h_{i1l_{i1}q_{i1l_{i1}}}) = u_{i1l_{i1}q_{i1l_{i1}}}$$

$$\vdots$$

$$F(p_{im_i 1}, h_{im_i 11}) = u_{im_i 11}$$

$$\vdots$$

$$F(p_{im_i 1}, h_{im_i 1q_{im_i 1}}) = u_{im_i 1q_{im_i 1}}$$

$$\vdots$$

$$F(p_{im_i l_{im_i}}, h_{im_i l_{im_i} 1}) = u_{im_i l_{im_i} 1}$$

$$\vdots$$

$$F(p_{im_i l_{im_i}}, h_{im_i l_{im_i} q_{im_i l_{im_i}}}) = u_{im_i l_{im_i} q_{im_i l_{im_i}}}$$

$$B(h_{i111}, p_{i11}) = \text{any, if } \neg Consumable(h_{i111})$$

$$\vdots$$

$$B(h_{i11q_{i11}}, p_{i11}) = \text{any, if } \neg Consumable(h_{i11q_{i11}})$$

$$\vdots$$

$$B(h_{im_i l_{im_i} 1}, p_{im_i l_{im_i}}) = \text{any, if } \neg Consumable(h_{im_i l_{im_i} 1})$$

$$\vdots$$

$$B(h_{im_i l_{im_i} q_{im_i l_{im_i}}}, p_{im_i l_{im_i}}) = \text{any, if } \neg Consumable(h_{im_i l_{im_i} q_{im_i l_{im_i}}})$$

If nothing is produced for a particular combination of reads (case P_0), only the equations that involve $b_{i,working}$ and $b_{i,idle}$ are defined in that case.

The queue alphabet of the FIFO Petri Net is defined as follows:

$$Q_{FPN} = OBJECTS_{RDFD} \cup \{I, W\}$$

The initial marking $M_{0,FPN}$ is such that:

$$M_{0,FPN}(b_{i,idle}) = I \qquad \forall i \in \{1,\ldots,b\}$$

$$M_{0,FPN}(b_{i,working:j}) =<> \qquad \forall i \in \{1,\ldots,b\}\ \forall j \in \{1,\ldots,m_i\}$$

$$M_{0,FPN}(f_j) = fs(f_j) \qquad \forall j \in \{1,\ldots,f\}, f_j \in F_{RDFD}$$

Then, the computation system $S_{FPN} = (T_{FPN}, M_{FPN}, M_{0,FPN}, \overset{-FPN}{\longrightarrow})$ is realized by the FIFO Petri Net FPN. Consider the homomorphism $h = (\tau, \rho) : S_{RDFD} \rightarrow S_{FPN}$ where the homomorphism $\tau : (B_{RDFD}, \{C, P\}, I\!N)^* \rightarrow T_{FPN}^*$ and the injection $\rho : (bm, r, fs) \rightarrow M_{FPN}$ are defined as follows:

$$\tau((b_i, a, n)) = \begin{cases} c_{in}, & \text{if } a = C \\ p_{in}, & \text{if } a = P \text{ and } Produce(b_i) \text{ is of type } P_1 \\ p_{in_0 n_1}, & \text{if } a = P, Produce(b_i) \text{ is of type } P_2, \text{ and } n = \sum_{j=1}^{n_0 - 1} l_{ij} + n_1 \end{cases}$$

$\rho(bm, r, fs)$ is such that $\forall i \in \{1,\ldots,b\}\ \forall j \in \{1,\ldots,m_i\}$

$$M_{FPN}(b_{i,idle}) = \begin{cases} I, & \text{if } bm(b_i) = idle \\ <>, & \text{if } bm(b_i) = working \end{cases}$$

$$M_{FPN}(b_{i,working:j}) = \begin{cases} <>, & \text{if } bm(b_i) = idle \\ W, & \text{if } bm(b_i) = working, (Consume(b_i))_j \text{ has been executed} \\ <>, & \text{if } bm(b_i) = working, \exists k, k \neq j : (Consume(b_i))_k \text{ has been executed} \end{cases}$$

and $\forall j \in \{1,\ldots,f\}. f_j \in F_{RDFD}$

$$M_{FPN}(f_j) = fs(f_j).$$

Obviously, τ and ρ are bijective. Also, h satisfies (2.2.4.4.1) and (2.2.4.9.1). Hence, h is an isomorphism by Theorem (2.2.4.11). ■

According to the Definitions and Theorems from [KM82], here summarized in Subsection 2.2.4, the isomorphism h constructed in the previous proof is also a bijective (hence injective, surjective, hence spanning), length preserving, and principal homomorphism.

Example (2.3.1.2): We will provide a small example how to construct a FIFO Petri Net for a given nf-RDFD. Our nf-RDFD only consists of three bubbles A, B, and C, two consumable flows f and *out*, and two persistent flows g and h (see Figure 2.1). Bubble A only produces combinations of X

on flow f and 0 on flow g or Y on flow f and 1 on flow g. Bubble B can forward any value on flow h if it reads a 1 on flow g. It must forward the same value if it reads a 0 on flow g. Therefore, bubble C can read all possible combinations $X/0$, $X/1$, $Y/0$, and $Y/1$ on its flows f and h. Initially, flows g and h both contain a 0. All other flows are empty.

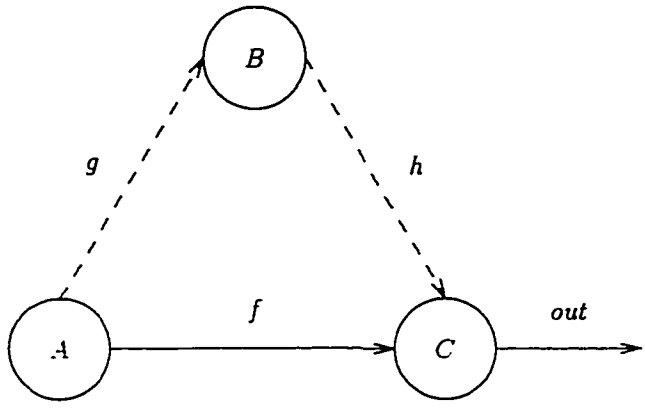

Figure 2.1: nf–RDFD.

The mappings *Enabled*, *Consume*, and *Produce* for the nf–RDFD shown in Figure 2.1 are specified as follows:

$$Enabled(A) = \lambda fs\ .\ true$$

$$Enabled(B) = \lambda fs\ .$$
$$Head(fs(g)) = 0 \vee Head(fs(g)) = 1$$

$$Enabled(C) = \lambda fs\ .$$
$$(\neg IsEmpty(f) \wedge Head(fs(f)) = X \wedge Head(fs(h)) = 0)$$
$$\vee(\neg IsEmpty(f) \wedge Head(fs(f)) = X \wedge Head(fs(h)) = 1)$$
$$\vee(\neg IsEmpty(f) \wedge Head(fs(f)) = Y \wedge Head(fs(h)) = 0)$$
$$\vee(\neg IsEmpty(f) \wedge Head(fs(f)) = Y \wedge Head(fs(h)) = 1)$$

$$Consume(A) = \lambda(fs, r)\ .\ \{(fs, r)\}$$

$$Consume(B) = \lambda(fs, r)\ .$$
$$\{\textbf{if } Head(fs(g)) = 0$$
$$\textbf{then } In(g, B)(fs, r)$$
$$\textbf{fi}.$$
$$\textbf{if } Head(fs(g)) = 1$$

then $In(g, B)(fs, r)$

fi

}

$Consume(C) = \lambda(fs, r)$.

{**if** $Head(fs(f)) = X \wedge Head(fs(h)) = 0$

then $In(f, C)(In(h, C)(fs, r))$

fi,

if $Head(fs(f)) = X \wedge Head(fs(h)) = 1$

then $In(f, C)(In(h, C)(fs, r))$

fi,

if $Head(fs(f)) = Y \wedge Head(fs(h)) = 0$

then $In(f, C)(In(h, C)(fs, r))$

fi,

if $Head(fs(f)) = Y \wedge Head(fs(h)) = 1$

then $In(f, C)(In(h, C)(fs, r))$

fi

}

$Produce(A) = \lambda(fs, r)$.

{$Out(X, f, A)(Out(0, g, A)(fs, r))$,

$Out(Y, f, A)(Out(1, g, A)(fs, r))$}

}

$Produce(B) = \lambda(fs, r)$.

{**if** $r(B)(g) = 0$

then $Out(0, h, B)(fs, r)$

fi,

if $r(B)(g) = 1$

then $Out(0, h, B)(fs, r)$

$\square\, Out(1, h, B)(fs, r)$

fi

}

$Produce(C) = \lambda(fs, r)$.

{**if** $r(C)(f) = X \wedge r(C)(h) = 0$
then $Out(X0, out, C)(fs, r)$
fi,
if $r(C)(f) = X \wedge r(C)(h) = 1$
then $Out(X1, out, C)(fs, r)$
fi,
if $r(C)(f) = Y \wedge r(C)(h) = 0$
then $Out(Y0, out, C)(fs, r)$
fi,
if $r(C)(f) = Y \wedge r(C)(h) = 1$
then $Out(Y1, out, C)(fs, r)$
fi
}

According to Theorem (2.3.1.1), the given nf–RDFD transforms into the following marked FIFO Petri Net $FPN = ((P_{FPN}, T_{FPN}, B_{FPN}, F_{FPN}, Q_{FPN}), M_{0,FPN})$:

$$P_{FPN} = \{f, g, h, out\} \cup \{A_i, A_{w:1}, B_i, B_{w:1}, B_{w:2}, C_i, C_{w:1}, C_{w:2}, C_{w:3}, C_{w:4}\}$$

$$T_{FPN} = \{C_{A1}, C_{B1}, C_{B2}, C_{C1}, C_{C2}, C_{C3}, C_{C4}\}$$
$$\cup\{P_{A1}, P_{A2}, P_{B11}, P_{B21}, P_{B22}, P_{C11}, P_{C21}, P_{C31}, P_{C41}\}$$

The initial marking $M_{0,FPN}$ is such that:

$$M_{0,FPN}(A_i) = M_{0,FPN}(B_i) = M_{0,FPN}(C_i) = I$$

$$M_{0,FPN}(A_{w:1}) = M_{0,FPN}(B_{w:1}) = M_{0,FPN}(B_{w:2}) =<>$$

$$M_{0,FPN}(C_{w:1}) = M_{0,FPN}(C_{w:2}) = M_{0,FPN}(B_{w:3}) = M_{0,FPN}(B_{w:4}) =<>$$

$$M_{0,FPN}(f) =<>$$

$$M_{0,FPN}(g) = 0$$

$$M_{0,FPN}(h) = 0$$

$$M_{0,FPN}(out) =<>$$

B_{FPN}, F_{FPN}, and Q_{FPN} can be gained from Figure 2.2. In this figure, we use a double arrow to indicate that the forward and backward incidence mappings for a particular place/transition combination are identical. i. e., $B(g, C_{B1}) = F(C_{B1}, g) = 0$. This means, that transition C_{B1} (if fired) removes the head element (i. e., 0) of place g and appends the same element to this place. Obviously, place g always contains only one element since it represents a persistent flow of the nf–RDFD. ■

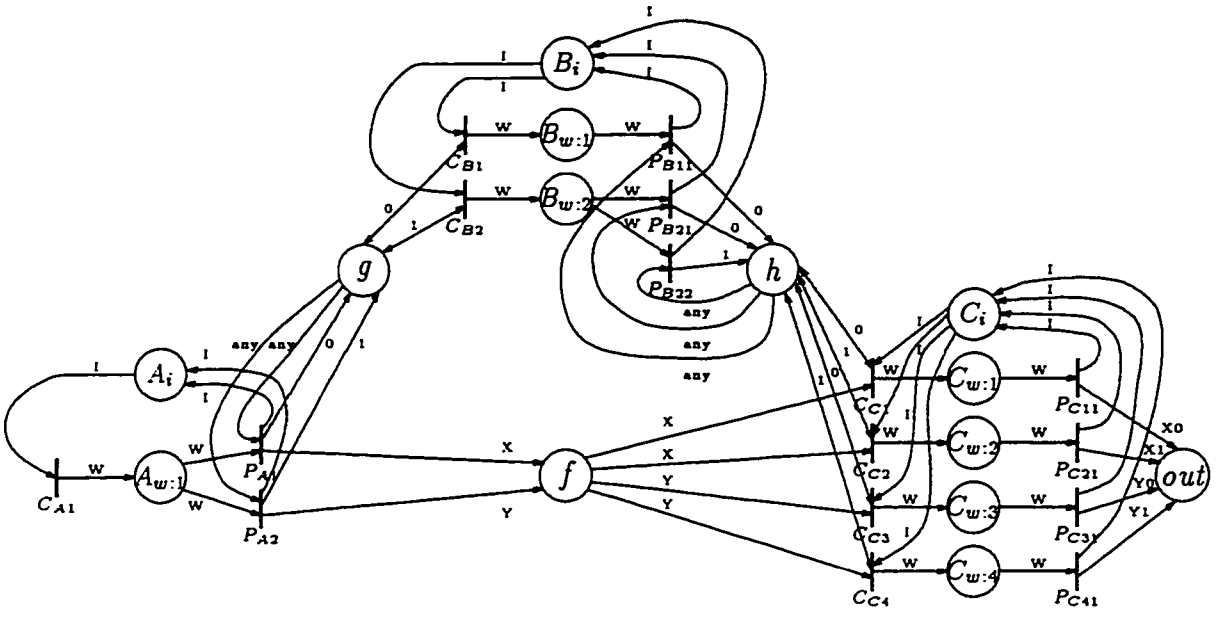

Figure 2.2: FIFO Petri Net.

2.3.2 Simulation of Program Machines by PFF–RDFD's

Theorem (2.3.2.1): Every Program Machine can be simulated by an PFF–RDFD.

Proof: Assume we have a Program Machine with registers $R = \{r_1, \ldots, r_p\}$, labels $Q = \{q_0, \ldots, q_r\}$ and instructions I. We will partially follow the proof of how to simulate a Program Machine by a FIFO Petri Net (given in [FM82] and [MF85]) when we indicate how to simulate a Program Machine by a PFF–RDFD.

We define our PFF–RDFD $RDFD_{PM}$, given in the normal form of Definition (2.2.1.7), as follows:

$$RDFD_{PM} = (B_{RDFD}, FLOWNAMES_{RDFD}, TYPES_{RDFD}, P_{RDFD}, F_{RDFD}),$$

where

$B_{RDFD} = \{q_0, \ldots, q_r\} \cup \{r_1, \ldots, r_p\}$

$FLOWNAMES_{RDFD} = \{do\_q_0, do\_q_0\_q_1\}$

$$\cup \bigcup_{s \in I} \{do\_q_s\_q_m \mid \text{instruction labeled } q_s \text{ is an increment instruction}\}$$

$$\cup \bigcup_{s \in I} \{do\_q_s\_q_m, do\_q_s\_q_l \mid \text{instruction labeled } q_s \text{ is a test and decrement instruction}\}$$

$$\cup \bigcup_{s \in I} \{i\_q_s\_r_i, o\_r_i\_q_s \mid \text{instruction labeled } q_s \text{ accesses register } r_i\}$$

$$\cup \bigcup_{i=1,\ldots,r} \{next_i, act_i, val_i\}$$

$TYPES_{RDFD}$ = ENAB ∪ ACTION ∪ RESULT ∪ FROM ∪ YES ∪ BIT

where ENAB = $\{start, go\}$, ACTION = $\{add, sub\}$, RESULT = $\{done, iszero\}$,

FROM = $\{1, \ldots, r\}$, YES = $\{yes\}$, and BIT = $\{0, 1\}$

$P_{RDFD} = \{consumable\}$

$F_{RDFD} = \{(q_0, do\_q_0, \text{ENAB}, q_0, consumable), (q_0, do\_q_0\_q_1, \text{ENAB}, q_1, consumable)\}$

$\cup\{(q_s, do\_q_s\_q_m, \text{ENAB}, q_m, consumable) \mid \forall\, FLOWNAMES\ do\_q_s\_q_m\}$

$\cup\{(q_s, i\_q_s\_r_i, \text{ACTION}, r_i, consumable) \mid \forall\, FLOWNAMES\ i\_q_s\_r_i\}$

$\cup\{(r_i, o\_r_i\_q_s, \text{RESULT}, q_s, consumable) \mid \forall\, FLOWNAMES\ o\_r_i\_q_s\}$

$$\cup \bigcup_{i=1,\ldots,r} \{(r_i, next_i, \text{FROM}, r_i, consumable),$$

$$(r_i, act_i, \text{YES}, r_i, consumable),$$

$$(r_i, val_i, \text{BIT}, r_i, consumable)\}$$

We use a unary representation for nonnegative integers which are the only possible contents of the registers of the simulated Program Machine. The value 0 stands for 0 and the sequence of values $(\underbrace{1, \ldots, 1}_{n}, 0)$ represents n. Initially, the values on the flows are *start* on do_q_0 and the unary representation of the initial contents of the registers $r_1, \ldots, r_p$, i. e., the values $k_1^1, \ldots, k_1^p$, on the flows $val_1, \ldots, val_p$, respectively. All other flows are empty.

The mapping *Enabled* is defined as follows:

If the bubble represents the start instruction (see Figure 2.3):

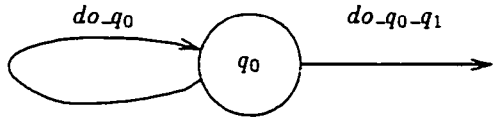

Figure 2.3: Bubble q_0.

$$Enabled(q_0) = \lambda fs\ .\ (\neg IsEmpty(do\_q_0) \wedge Head(fs(do\_q_0)) = start)$$

If the instruction labeled q_s is an increment instruction accessing register r_i and its corresponding bubble has inflows $do\_q_{m_1}\_q_s, \ldots, do\_q_{m_s}\_q_s, o\_r_i\_q_s$ (see Figure 2.4):

$$Enabled(q_s) = \lambda fs\ .$$

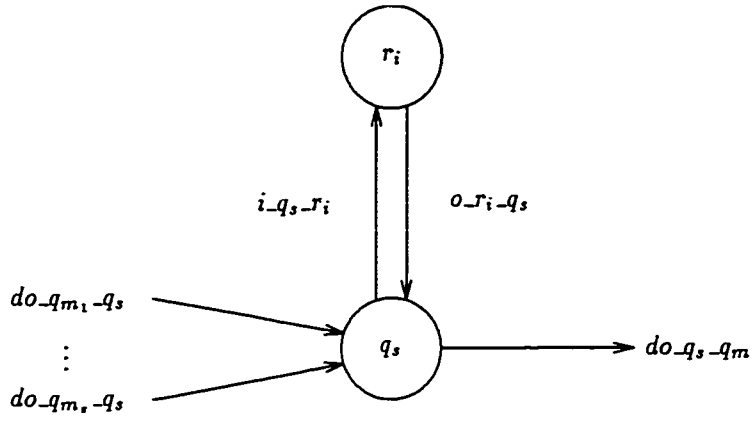

Figure 2.4: Bubble q_s for Increment Instruction.

$$(\neg IsEmpty(do\_q_{m_1}\_q_s) \wedge Head(fs(do\_q_{m_1}\_q_s)) = go)$$
$$\vee \ldots \vee$$
$$(\neg IsEmpty(do\_q_{m_t}\_q_s) \wedge Head(fs(do\_q_{m_t}\_q_s)) = go)$$
$$\vee(\neg IsEmpty(o\_r_i\_q_s) \wedge Head(fs(o\_r_i\_q_s)) = done)$$

If the instruction labeled q_s is a test and decrement instruction accessing register r_i and its corresponding bubble has inflows $do\_q_{m_1}\_q_s, \ldots, do\_q_{m_t}\_q_s, o\_r_i\_q_s$ (see Figure 2.5):

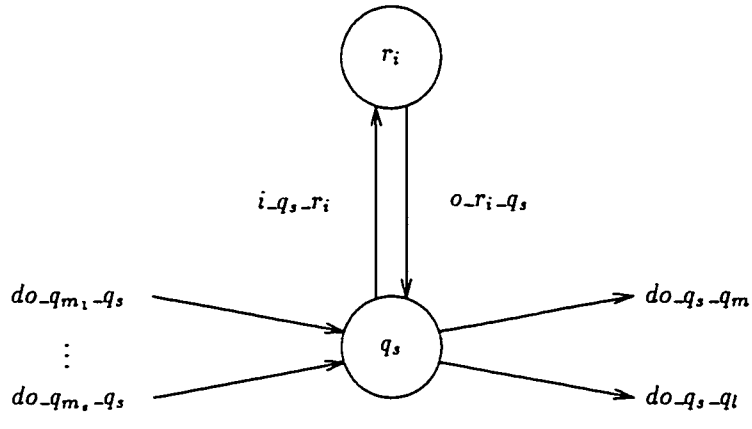

Figure 2.5: Bubble q_s for Test and Decrement Instruction.

$$Enabled(q_s) = \lambda fs\ .$$
$$(\neg IsEmpty(do\_q_{m_1}\_q_s) \wedge Head(fs(do\_q_{m_1}\_q_s)) = go)$$
$$\vee \ldots \vee$$
$$(\neg IsEmpty(do\_q_{m_t}\_q_s) \wedge Head(fs(do\_q_{m_t}\_q_s)) = go)$$

$\vee(\neg IsEmpty(o\_r_i\_q_s) \wedge Head(fs(o\_r_i\_q_s)) = done)$

$\vee(\neg IsEmpty(o\_r_i\_q_s) \wedge Head(fs(o\_r_i\_q_s)) = iszero)$

If the instruction labeled q_s is the halt instruction, i. e., $q_s = q_f$, and its corresponding bubble has inflows $do\_q_{m_1}\_q_f, \ldots, do\_q_{m_f}\_q_f$ (see Figure 2.6):

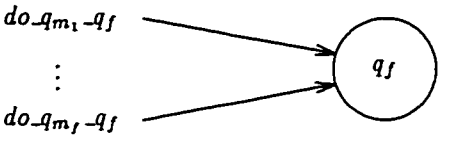

Figure 2.6: Bubble q_f.

$Enabled(q_f) = \lambda fs\ .\ false$

If the bubble representing register r_i has inflows $i\_q_{s_1}\_r_i, \ldots, i\_q_{s_t}\_r_i, next_i, act_i, val_i$ (see Figure 2.7):

$Enabled(r_i) = \lambda fs\ .$

[These lines are required for add:]

$(\neg IsEmpty(i\_q_{s_1}\_r_i) \wedge Head(fs(i\_q_{s_1}\_r_i)) = add)$

$\vee \ldots \vee$

$(\neg IsEmpty(i\_q_{s_t}\_r_i) \wedge Head(fs(i\_q_{s_t}\_r_i)) = add)$

[These lines are required for sub:]

$\vee(\neg IsEmpty(i\_q_{s_1}\_r_i) \wedge Head(fs(i\_q_{s_1}\_r_i)) = sub$

$\quad \wedge \neg IsEmpty(val_i) \wedge Head(fs(val_i)) = 0)$

$\vee \ldots \vee$

$(\neg IsEmpty(i\_q_{s_t}\_r_i) \wedge Head(fs(i\_q_{s_t}\_r_i)) = sub$

$\quad \wedge \neg IsEmpty(val_i) \wedge Head(fs(val_i)) = 0)$

$\vee(\neg IsEmpty(i\_q_{s_1}\_r_i) \wedge Head(fs(i\_q_{s_1}\_r_i)) = sub$

$\quad \wedge \neg IsEmpty(val_i) \wedge Head(fs(val_i)) = 1)$

$\vee \ldots \vee$

$(\neg IsEmpty(i\_q_{s_t}\_r_i) \wedge Head(fs(i\_q_{s_t}\_r_i)) = sub$

$\quad \wedge \neg IsEmpty(val_i) \wedge Head(fs(val_i)) = 1)$

[This line is required to copy the tail of 1's:]

$\vee(\neg IsEmpty(act_i) \wedge Head(fs(act_i)) = yes$

$\quad \wedge \neg IsEmpty(val_i) \wedge Head(fs(val_i)) = 1)$

[These lines are required to activate the next instruction:]

$$\vee(\neg IsEmpty(act_i) \wedge Head(fs(act_i)) = yes$$
$$\wedge\neg IsEmpty(val_i) \wedge Head(fs(val_i)) = 0$$
$$\wedge\neg IsEmpty(next_i) \wedge Head(fs(next_i)) = s_1)$$
$$\vee \ldots \vee$$
$$(\neg IsEmpty(act_i) \wedge Head(fs(act_i)) = yes$$
$$\wedge\neg IsEmpty(val_i) \wedge Head(fs(val_i)) = 0$$
$$\wedge\neg IsEmpty(next_i) \wedge Head(fs(next_i)) = s_i)$$

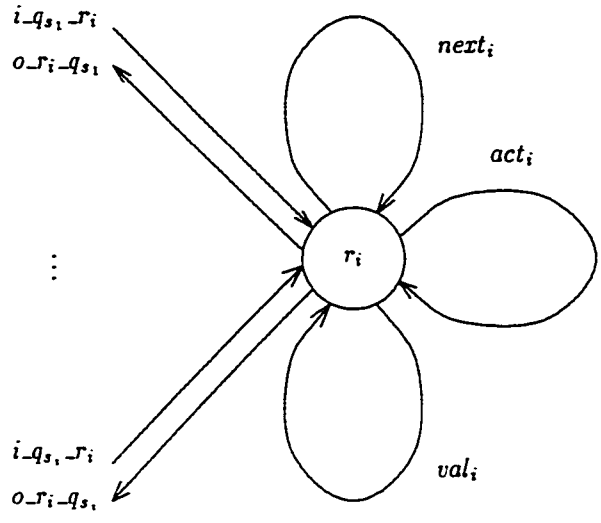

Figure 2.7: Bubble r_i.

As usual, we assume that whenever a set of input conditions on flows enables a bubble, all the head elements on these flows will be consumed when this bubble actually goes from *idle* to *working*. Therefore, we omit to list the mapping *Consume*. We immediately continue with *Produce*.

If the bubble represents the start instruction (see Figure 2.3):

$Produce(q_0) = \lambda(fs, r)$.

{**if** $r(q_0)(do\_q_0) = start$

then $Out(go, do\_q_0\_q_1, q_0)(fs, r)$

fi

}

If the instruction labeled q_s is an increment instruction accessing register r_i and its corresponding bubble has inflows $do\_q_{m_1}\_q_s, \ldots, do\_q_{m_r}\_q_s, o\_r_i\_q_s$ and outflows $do\_q_s\_q_m, i\_q_s\_r_i$ (see Figure 2.4):

$Produce(q_s) = \lambda(fs, r)$.

{**if** $r(q_s)(do\_q_{m_1}\_q_s) = go$

then $Out(add, i\_q_s\_r_i, q_s)(fs, r)$

fi,

...,

if $r(q_s)(do\_q_{m_t}\_q_s) = go$

then $Out(add, i\_q_s\_r_i, q_s)(fs, r)$

fi,

if $r(q_s)(o\_r_i\_q_s) = done$

then $Out(go, do\_q_s\_q_m, q_s)(fs, r)$

fi

}

If the instruction labeled q_s is a test and decrement instruction accessing register r_i and its corresponding bubble has inflows $do\_q_{m_1}\_q_s, \ldots, do\_q_{m_t}\_q_s, o\_r_i\_q_s$ and outflows $do\_q_s\_q_m, do\_q_s\_q_l, i\_q_s\_r_i$ (see Figure 2.5):

$Produce(q_s) = \lambda(fs, r)$.

{**if** $r(q_s)(do\_q_{m_1}\_q_s) = go$

then $Out(sub, i\_q_s\_r_i, q_s)(fs, r)$

fi,

...,

if $r(q_s)(do\_q_{m_t}\_q_s) = go$

then $Out(sub, i\_q_s\_r_i, q_s)(fs, r)$

fi,

if $r(q_s)(o\_r_i\_q_s) = done$

then $Out(go, do\_q_s\_q_m, q_s)(fs, r)$

fi,

if $r(q_s)(o\_r_i\_q_s) = iszero$

then $Out(go, do\_q_s\_q_l, q_s)(fs, r)$

fi

}

The next line is given only for formal reasons, but it will never be executed since the corresponding enabling condition is always *false*. If the instruction labeled q_s is the halt instruction, i. e., $q_s = q_f$,

and its corresponding bubble has inflows $do\_q_{m_1}\_q_f, \ldots, do\_q_{m_f}\_q_f$ (see Figure 2.6):

$$Produce(q_f) = \lambda(fs, r) \,.\, \{(fs, [q_f \mapsto \lambda f \,.\, \bot]r)\}$$

If the bubble representing register r_i has inflows $i\_q_{s_1}\_r_i, \ldots, i\_q_{s_t}\_r_i, next_i, act_i, val_i$ and outflows $o\_r_i\_q_{s_1}, \ldots, o\_r_i\_q_{s_t}, next_i, act_i, val_i$ (see Figure 2.7):

$Produce(r_i) = \lambda(fs, r)$.

[These lines are required for add:]

$\{$**if** $r(r_i)(i\_q_{s_1}\_r_i) = add$

then $Out(s_1, next_i, r_i)(Out(yes, act_i, r_i)(Out(1, val_i, r_i)(fs, r)))$

fi,

$\ldots$,

if $r(r_i)(i\_q_{s_t}\_r_i) = add$

then $Out(s_i, next_i, r_i)(Out(yes, act_i, r_i)(Out(1, val_i, r_i)(fs, r)))$

fi,

[These lines are required for sub:]

if $r(r_i)(i\_q_{s_1}\_r_i) = sub \wedge r(r_i)(val_i) = 0$

then $Out(0, val_i, r_i)(Out(iszero, o\_r_i\_q_{s_1}, r_i)(fs, r))$

fi,

$\ldots$,

if $r(r_i)(i\_q_{s_t}\_r_i) = sub \wedge r(r_i)(val_i) = 0$

then $Out(0, val_i, r_i)(Out(iszero, o\_r_i\_q_{s_t}, r_i)(fs, r))$

fi,

if $r(r_i)(i\_q_{s_1}\_r_i) = sub \wedge r(r_i)(val_i) = 1$

then $Out(s_1, next_i, r_i)(Out(yes, act_i, r_i)(fs, r))$

fi.

$\ldots$,

if $r(r_i)(i\_q_{s_t}\_r_i) = sub \wedge r(r_i)(val_i) = 1$

then $Out(s_i, next_i, r_i)(Out(yes, act_i, r_i)(fs, r))$

fi.

[This line is required to copy the tail of 1's:]

if $r(r_i)(act_i) = yes \wedge r(r_i)(val_i) = 1$

then $Out(yes, act_i, r_i)(Out(1, val_i, r_i)(fs, r))$

fi,

[These lines are required to activate the next instruction:]

if $r(r_i)(act_i) = yes \wedge r(r_i)(val_i) = 0 \wedge r(r_i)(next_i) = s_1$

then $Out(0, val_i, r_i)(Out(done, o\_r_i\_q_{s_1}, r_i)(fs, r))$

fi,

$\ldots$,

if $r(r_i)(act_i) = yes \wedge r(r_i)(val_i) = 0 \wedge r(r_i)(next_i) = s_i$

then $Out(0, val_i, r_i)(done, Out(o\_r_i\_q_{s_i}, r_i)(fs, r))$

fi

}

■

Example (2.3.2.2): Consider a Program Machine that initially has some nonnegative values in its registers r_1 and r_2, while r_3 contains 0. The following code stores the sum of r_1 and r_2 in r_3 and erases the original values:

q_0 : **start goto** q_1;

q_1 : **if** $r_1 = 0$ **then goto** q_3 **else** $r_1 := r_1 - 1$ **goto** q_2;

q_2 : $r_3 := r_3 + 1$ **goto** q_1;

q_3 : **if** $r_2 = 0$ **then goto** q_f **else** $r_2 := r_2 - 1$ **goto** q_4;

q_4 : $r_3 := r_3 + 1$ **goto** q_3;

q_f : **halt.**

The graphical representation of the equivalent PFF–RDFD is given in Figure 2.8.

Based on this diagram, the formal definitions can be gained according to Theorem (2.3.2.1) as:

$B_{RDFD} = \{q_0, q_1, q_2, q_3, q_4, q_f\} \cup \{r_1, r_2, r_3\}$

$FLOWNAMES_{RDFD} = \{do\_q_0, do\_q_0\_q_1\}$

$\cup\{do\_q_2\_q_1, do\_q_4\_q_3\}$

$\cup\{do\_q_1\_q_3, do\_q_1\_q_2, do\_q_3\_q_f, do\_q_3\_q_4\}$

$\cup\{i\_q_1\_r_1, o\_r_1\_q_1, i\_q_2\_r_3, o\_r_3\_q_2, i\_q_3\_r_2, o\_r_2\_q_3, i\_q_4\_r_3, o\_r_3\_q_4\}$

$\cup\{next_1, act_1, val_1, next_2, act_2, val_2, next_3, act_3, val_3\}$

$TYPES_{RDFD}$ (with FROM $= \{1, 2, 3, 4\}$). P_{RDFD}, and F_{RDFD} follow immediately.

Assume we want to add 2 and 3, then the initial values, i. e., sequences of values, on the flows are $do\_q_0 = start$, $val_1 = (1, 1, 0)$, $val_2 = (1, 1, 1, 0)$ and $val_3 = 0$. All other flows are empty.

The mapping *Enabled* is defined as follows:

$Enabled(q_0) = \lambda fs$.

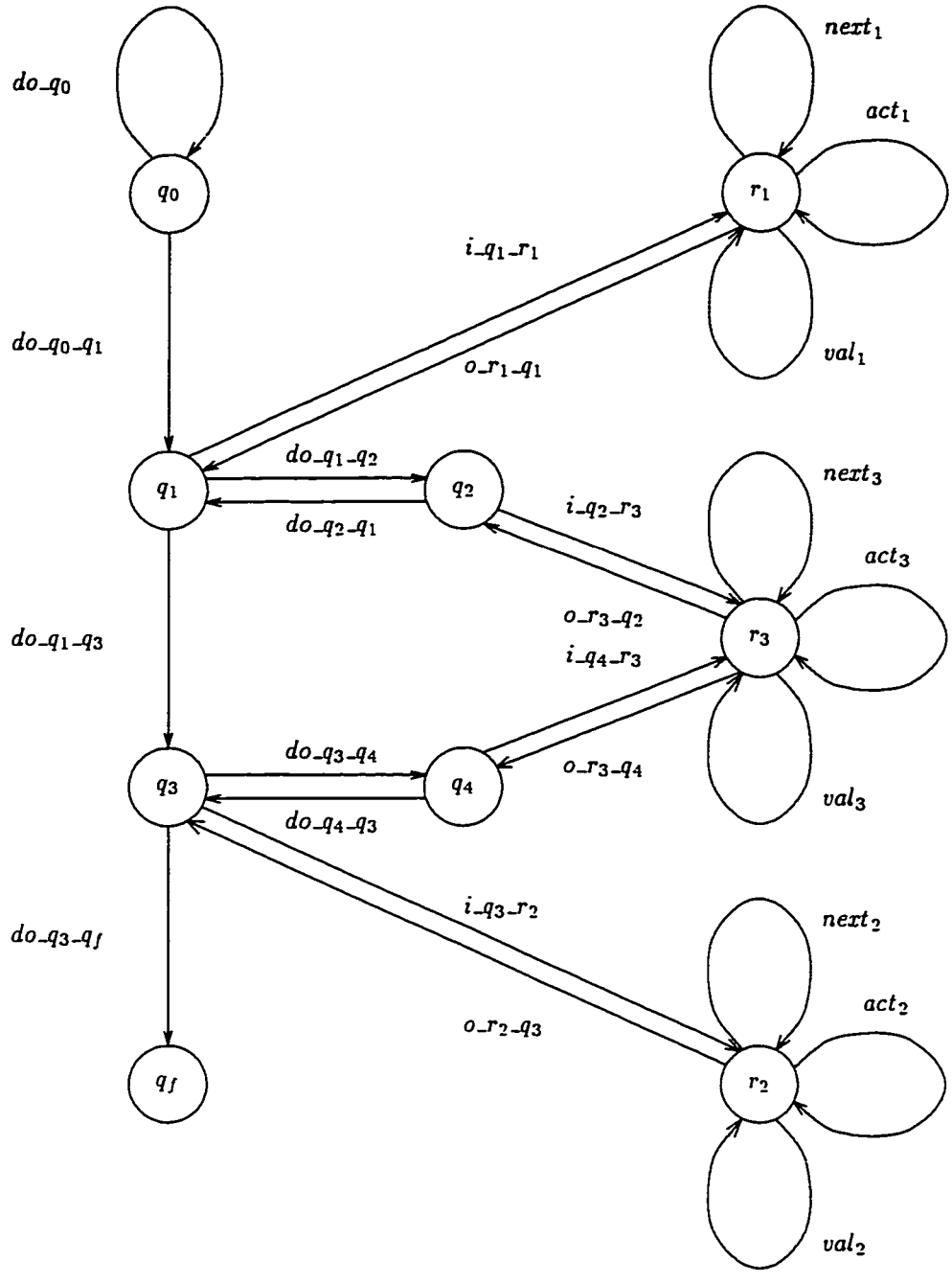

Figure 2.8: PFF–RDFD.

$$(\neg IsEmpty(do\_q_0) \wedge Head(fs(do\_q_0)) = start)$$

$$Enabled(q_1) = \lambda fs\ .$$

$$(\neg IsEmpty(do\_q_0\_q_1) \wedge Head(fs(do\_q_0\_q_1)) = go)$$

$$\vee(\neg IsEmpty(do\_q_2\_q_1) \wedge Head(fs(do\_q_2\_q_1)) = go)$$

$$\vee(\neg IsEmpty(o\_r_1\_q_1) \wedge Head(fs(o\_r_1\_q_1)) = done)$$

$$\vee(\neg IsEmpty(o\_r_1\_q_1) \wedge Head(fs(o\_r_1\_q_1)) = iszero)$$

$Enabled(q_2) = \lambda fs\ .$

$$(\neg IsEmpty(do\_q_1\_q_2) \wedge Head(fs(do\_q_1\_q_2)) = go)$$

$$\vee(\neg IsEmpty(o\_r_3\_q_2) \wedge Head(fs(o\_r_3\_q_2)) = done)$$

$Enabled(q_3) = \lambda fs\ .$

$$(\neg IsEmpty(do\_q_1\_q_3) \wedge Head(fs(do\_q_1\_q_3)) = go)$$

$$\vee(\neg IsEmpty(do\_q_4\_q_3) \wedge Head(fs(do\_q_4\_q_3)) = go)$$

$$\vee(\neg IsEmpty(o\_r_2\_q_3) \wedge Head(fs(o\_r_2\_q_3)) = done)$$

$$\vee(\neg IsEmpty(o\_r_2\_q_3) \wedge Head(fs(o\_r_2\_q_3)) = iszero)$$

$Enabled(q_4) = \lambda fs\ .$

$$(\neg IsEmpty(do\_q_3\_q_4) \wedge Head(fs(do\_q_3\_q_4)) = go)$$

$$\vee(\neg IsEmpty(o\_r_3\_q_4) \wedge Head(fs(o\_r_3\_q_4)) = done)$$

$Enabled(q_f) = \lambda fs\ .\ false$

$Enabled(r_1) = \lambda fs\ .$

$$(\neg IsEmpty(i\_q_1\_r_1) \wedge Head(fs(i\_q_1\_r_1)) = add)$$

$$\vee(\neg IsEmpty(i\_q_1\_r_1) \wedge Head(fs(i\_q_1\_r_1)) = sub$$

$$\wedge\neg IsEmpty(val_1) \wedge Head(fs(val_1)) = 0)$$

$$\vee(\neg IsEmpty(i\_q_1\_r_1) \wedge Head(fs(i\_q_1\_r_1)) = sub$$

$$\wedge\neg IsEmpty(val_1) \wedge Head(fs(val_1)) = 1)$$

$$\vee(\neg IsEmpty(act_1) \wedge Head(fs(act_1)) = yes$$

$$\wedge\neg IsEmpty(val_1) \wedge Head(fs(val_1)) = 1)$$

$$\vee(\neg IsEmpty(act_1) \wedge Head(fs(act_1)) = yes$$

$$\wedge\neg IsEmpty(val_1) \wedge Head(fs(val_1)) = 0$$

$$\wedge\neg IsEmpty(next_1) \wedge Head(fs(next_1)) = 1)$$

$Enabled(r_2) = \lambda fs\ .$

$$(\neg IsEmpty(i\_q_3\_r_2) \wedge Head(fs(i\_q_3\_r_2)) = add)$$

$$\vee(\neg IsEmpty(i\_q_3\_r_2) \wedge Head(fs(i\_q_3\_r_2)) = sub$$

$$\wedge\neg IsEmpty(val_2) \wedge Head(fs(val_2)) = 0)$$

$$\vee(\neg IsEmpty(i\_q_3\_r_2) \wedge Head(fs(i\_q_3\_r_2)) = sub$$

$$\wedge\neg IsEmpty(val_2) \wedge Head(fs(val_2)) = 1)$$

$$\vee(\neg IsEmpty(act_2) \wedge Head(fs(act_2)) = yes$$

$$\wedge \neg IsEmpty(val_2) \wedge Head(fs(val_2)) = 1)$$

$$\vee(\neg IsEmpty(act_2) \wedge Head(fs(act_2)) = yes$$

$$\wedge \neg IsEmpty(val_2) \wedge Head(fs(val_2)) = 0$$

$$\wedge \neg IsEmpty(next_2) \wedge Head(fs(next_2)) = 3)$$

$$Enabled(r_3) = \lambda fs\ .$$

$$(\neg IsEmpty(i\_q_2\_r_3) \wedge Head(fs(i\_q_2\_r_3)) = add)$$

$$\vee(\neg IsEmpty(i\_q_4\_r_3) \wedge Head(fs(i\_q_4\_r_3)) = add)$$

$$\vee(\neg IsEmpty(i\_q_2\_r_3) \wedge Head(fs(i\_q_2\_r_3)) = sub$$

$$\wedge \neg IsEmpty(val_3) \wedge Head(fs(val_3)) = 0)$$

$$\vee(\neg IsEmpty(i\_q_4\_r_3) \wedge Head(fs(i\_q_4\_r_3)) = sub$$

$$\wedge \neg IsEmpty(val_3) \wedge Head(fs(val_3)) = 0)$$

$$\vee(\neg IsEmpty(i\_q_2\_r_3) \wedge Head(fs(i\_q_2\_r_3)) = sub$$

$$\wedge \neg IsEmpty(val_3) \wedge Head(fs(val_3)) = 1)$$

$$\vee(\neg IsEmpty(i\_q_4\_r_3) \wedge Head(fs(i\_q_4\_r_3)) = sub$$

$$\wedge \neg IsEmpty(val_3) \wedge Head(fs(val_3)) = 1)$$

$$\vee(\neg IsEmpty(act_3) \wedge Head(fs(act_3)) = yes$$

$$\wedge \neg IsEmpty(val_3) \wedge Head(fs(val_3)) = 1)$$

$$\vee(\neg IsEmpty(act_3) \wedge Head(fs(act_3)) = yes$$

$$\wedge \neg IsEmpty(val_3) \wedge Head(fs(val_3)) = 0$$

$$\wedge \neg IsEmpty(next_3) \wedge Head(fs(next_3)) = 2)$$

$$\vee(\neg IsEmpty(act_3) \wedge Head(fs(act_3)) = yes$$

$$\wedge \neg IsEmpty(val_3) \wedge Head(fs(val_3)) = 0$$

$$\wedge \neg IsEmpty(next_3) \wedge Head(fs(next_3)) = 4)$$

As usual, we assume that whenever a set of input conditions on flows enables a bubble, all the head elements on these flows will be consumed when this bubble actually goes from *idle* to *working*. Therefore, we omit to list the mapping *Consume*. We immediately continue with the mapping *Produce*.

$$Produce(q_0) = \lambda(fs, r)\ .$$

$$\{\textbf{if } r(q_0)(do\_q_0) = start$$

$$\textbf{then } Out(go, do\_q_0\_q_1, q_0)(fs, r)$$

$$\textbf{fi}$$

$$\}$$

$$Produce(q_1) = \lambda(fs, r)\ .$$

$\{$**if** $r(q_1)(do\_q_0\_q_1) = go$

then $Out(sub, i\_q_1\_r_1, q_1)(fs, r)$

fi,

if $r(q_1)(do\_q_2\_q_1) = go$

then $Out(sub, i\_q_1\_r_1, q_1)(fs, r)$

fi,

if $r(q_1)(o\_r_1\_q_1) = done$

then $Out(go, do\_q_1\_q_2, q_1)(fs, r)$

fi,

if $r(q_1)(o\_r_1\_q_1) = iszero$

then $Out(go, do\_q_1\_q_3, q_1)(fs, r)$

fi

$\}$

$Produce(q_2) = \lambda(fs, r)$.

$\{$**if** $r(q_2)(do\_q_1\_q_2)) = go$

then $Out(add, i\_q_2\_r_3, q_2)(fs, r)$

fi,

if $r(q_2)(o\_r_3\_q_2)) = done$

then $Out(go, do\_q_2\_q_1, q_2)(fs, r)$

fi

$\}$

$Produce(q_3) = \lambda(fs, r)$.

$\{$**if** $r(q_3)(do\_q_1\_q_3) = go$

then $Out(sub, i\_q_3\_r_2, q_3)(fs, r)$

fi,

if $r(q_3)(do\_q_4\_q_3) = go$

then $Out(sub, i\_q_3\_r_2, q_3)(fs, r)$

fi,

if $r(q_3)(o\_r_2\_q_3) = done$

then $Out(go, do\_q_3\_q_4, q_3)(fs, r)$

fi,

if $r(q_3)(o\_r_2\_q_3) = iszero$

then $Out(go, do\_q_3\_q_f, q_3)(fs, r)$

fi

}

$Produce(q_4) = \lambda(fs, r)$.

{**if** $r(q_4)(do\_q_3\_q_4)) = go$

then $Out(add, i\_q_4\_r_3, q_4)(fs, r)$

fi,

if $r(q_4)(o\_r_3\_q_4)) = done$

then $Out(go, do\_q_4\_q_3, q_4)(fs, r)$

fi

}

$Produce(q_f) = \lambda(fs, r) . \{(fs, [q_f \mapsto \lambda f . \perp]r)\}$

$Produce(r_1) = \lambda(fs, r)$.

{**if** $r(r_1)(i\_q_1\_r_1) = add$

then $Out(1, next_1, r_1)(Out(yes, act_1, r_1)(Out(1, val_1, r_1)(fs, r)))$

fi,

if $r(r_1)(i\_q_1\_r_1) = sub \wedge r(r_1)(val_1) = 0$

then $Out(0, val_1, r_1)(Out(iszero, o\_r_1\_q_1, r_1)(fs, r))$

fi,

if $r(r_1)(i\_q_1\_r_1) = sub \wedge r(r_1)(val_1) = 1$

then $Out(1, next_1, r_1)(Out(yes, act_1, r_1)(fs, r))$

fi,

if $r(r_1)(act_1) = yes \wedge r(r_1)(val_1) = 1$

then $Out(yes, act_1, r_1)(Out(1, val_1, r_1)(fs, r))$

fi,

if $r(r_1)(act_1) = yes \wedge r(r_1)(val_1) = 0 \wedge r(r_1)(next_1) = 1$

then $Out(0, val_1, r_1)(Out(done, o\_r_1\_q_1, r_1)(fs, r))$

fi

}

$Produce(r_2) = \lambda(fs, r)$.

{**if** $r(r_2)(i\_q_3\_r_2) = add$

then $Out(3, next_2, r_2)(Out(yes, act_2, r_2)(Out(1, val_2, r_2)(fs, r)))$

$$
\begin{aligned}
&\textbf{fi},\\
&\textbf{if } r(r_2)(i\_q_3\_r_2) = sub \wedge r(r_2)(val_2) = 0\\
&\textbf{then } Out(0, val_2, r_2)(Out(iszero, o\_r_2\_q_3, r_2)(fs, r))\\
&\textbf{fi},\\
&\textbf{if } r(r_2)(i\_q_3\_r_2) = sub \wedge r(r_2)(val_2) = 1\\
&\textbf{then } Out(3, next_2, r_2)(Out(yes, act_2, r_2)(fs, r))\\
&\textbf{fi},\\
&\textbf{if } r(r_2)(act_2) = yes \wedge r(r_2)(val_2) = 1\\
&\textbf{then } Out(yes, act_2, r_2)(Out(1, val_2, r_2)(fs, r))\\
&\textbf{fi},\\
&\textbf{if } r(r_2)(act_2) = yes \wedge r(r_2)(val_2) = 0 \wedge r(r_2)(next_2) = 3\\
&\textbf{then } Out(0, val_2, r_2)(Out(done, o\_r_2\_q_3, r_2)(fs, r))\\
&\textbf{fi}\\
&\}\\
Produce(r_3) = \lambda(fs, r)\ .&\\
&\{\textbf{if } r(r_3)(i\_q_2\_r_3) = add\\
&\textbf{then } Out(2, next_3, r_3)(Out(yes, act_3, r_3)(Out(1, val_3, r_3)(fs, r)))\\
&\textbf{fi},\\
&\textbf{if } r(r_3)(i\_q_4\_r_3) = add\\
&\textbf{then } Out(4, next_3, r_3)(Out(yes, act_3, r_3)(Out(1, val_3, r_3)(fs, r)))\\
&\textbf{fi},\\
&\textbf{if } r(r_3)(i\_q_2\_r_3) = sub \wedge r(r_3)(val_3) = 0\\
&\textbf{then } Out(0, val_3, r_3)(Out(iszero, o\_r_3\_q_2, r_3)(fs, r))\\
&\textbf{fi},\\
&\textbf{if } r(r_3)(i\_q_4\_r_3) = sub \wedge r(r_3)(val_3) = 0\\
&\textbf{then } Out(0, val_3, r_3)(Out(iszero, o\_r_3\_q_4, r_3)(fs, r))\\
&\textbf{fi},\\
&\textbf{if } r(r_3)(i\_q_2\_r_3) = sub \wedge r(r_3)(val_3) = 1\\
&\textbf{then } Out(2, next_3, r_3)(Out(yes, act_3, r_3)(fs, r))\\
&\textbf{fi},\\
&\textbf{if } r(r_3)(i\_q_4\_r_3) = sub \wedge r(r_3)(val_3) = 1\\
&\textbf{then } Out(4, next_3, r_3)(Out(yes, act_3, r_3)(fs, r))
\end{aligned}
$$

fi,

if $r(r_3)(act_3) = yes \wedge r(r_3)(val_3) = 1$

then $Out(yes, act_3, r_3)(Out(1, val_3, r_3)(fs, r))$

fi,

if $r(r_3)(act_3) = yes \wedge r(r_3)(val_3) = 0 \wedge r(r_3)(next_3) = 2$

then $Out(0, val_3, r_3)(Out(done, o\_r_3\_q_2, r_3)(fs, r))$

fi,

if $r(r_3)(act_3) = yes \wedge r(r_3)(val_3) = 0 \wedge r(r_3)(next_3) = 4$

then $Out(0, val_3, r_3)(Out(done, o\_r_3\_q_4, r_3)(fs, r))$

fi

}

■

2.4 Summary

In this paper, we have shown that PFF–RDFD's have the computational power of Turing Machines. Therefore, all interesting decidability problems such as reachability, termination, deadlock and liveness properties, and finiteness, that are undecidable for Turing Machines are undecidable for PFF–RDFD's, too.

Future work will have two directions: (i) simulation of FDFD's that make use of persistent flows, stores, infinite domains for flow values, and the facility for testing for empty flows through PFF–RDFD's and (ii) further restrictions on RDFD's. Direction (i) should help to provide a mechanism such that existing FDFD's can be transformed into a basic model. Then, such a model might be used as input to computer software for formal analysis and execution of FDFD's such as the ML interpreter described in [Wah95].

In direction (ii), we hope to find subclasses of RDFD's where some decidability problems can be solved. In particular, we would like to show that some of these subclasses can be simulated by Monogeneous FIFO Petri Nets, Linear FIFO Petri Nets, and Topologically Free Choice FIFO Petri Nets, respectively, and that this simulation is still based on an isomorphism. Then, since this type of a homomorphism preserves some decidability problems ([KM82]), we could immediately apply the results and algorithms known for a subclass of FIFO Petri Nets ([FM82], [MF85], [Fin86], [Rou87], [FC88], [FR88], to mention only a few) to the corresponding subclass of RDFD's.

Acknowledgements

Symanzik's research was partially supported by a German "DAAD–Doktorandenstipendium aus Mitteln des zweiten Hochschulsonderprogramms".

3 NON–ATOMIC COMPONENTS OF DATA FLOW DIAGRAMS: STORES, PERSISTENT FLOWS, AND TESTS FOR EMPTY FLOWS

A conference contribution submitted to STEP '97, London, UK (July 1997)

Jürgen Symanzik[1] and Albert L. Baker[2]

Abstract

It has been shown in [SB96a] that a particular subclass of Formalized Data Flow Diagrams (FDFD's) is Turing equivalent. We call this Turing equivalent subclass of FDFD's persistent flow–free Reduced Data Flow Diagrams (PFF–RDFD's). PFF–RDFD's do not contain persistent flows, reference only values whose types have finite domains, and have enabling conditions that contain no tests for empty flows. In addition, FDFD's do not contain (direct) representations of stores. This raises the question whether any of these common features of traditional Data Flow Diagrams elevates the expressive power of FDFD's, or whether the various subclasses have the same expressive power as FDFD's with these features. This paper addresses this issue of whether persistent flows, arbitrary domains, tests for empty flows or stores are essential features with respect to the expressive power of Formalized Data Flow Diagrams.

Keywords

Software Specifications, Data Flow Diagrams, Models of Computation.

---

[1] Primary researcher and author.
[2] Professor, Department of Computer Science, Iowa State University.

3.1 Introduction

Traditional Data Flow Diagrams (DFD's) are probably the most widely used specification technique in industry today. They are the cornerstone of the software development methodology commonly referred to as "Structured Analysis" (SA) ([You89]). Their popularity arises from their graphical representation and hierarchical structure, which allows users with non–technical backgrounds to work with them.

One of the drawbacks of traditional DFD's as a specification tool is that they have no rigorous or standard interpretation. In particular, traditional DFD's are usually considered static roadmaps of information flow in systems in which the *bubbles* — data transformers — of DFD's are the cities and *data flows* are the roads. Thus, a DFD can not specify system functionality, i. e., a DFD can not define the I/O behavior of a system. (This would require modelling of car movement in the roadmap analogy.) But defining system functionality is one of the things specifications should do.

Numerous formalizations of DFD's have appeared in the technical literature, e. g., in [DeM78], [WM85a], [WM85b], [Har87], [TP89], [You89], [Har92], and [Har96]. These attempts involve, in part, a more dynamic interpretation of data movement in DFD's. Given this type of rigorous and dynamic semantics, a DFD can serve as a formal specification of system functionality.

The authors use the approach to formalizing DFD's developed originally in [Col91], [CB94], [WBL93] and refined more precisely in [WBL93] and [LWBL96]. These Formalized Data Flow Diagrams (FDFD's) are described more fully in the next section. The formalization is based on defining bubble behavior in terms of *enabling conditions*, which define the pre–state required for a bubble to "do its thing", and *post–conditions*, which define a bubble's outputs in terms of its inputs. The enabling conditions and post–conditions are defined assertionally using *First Order Predicate Calculus* (FOPC) over abstract types, and are refered to as *firing rules*[3].

In other recent work the authors have shown that a restricted class of FDFD's (we call them PFF–RDFD's) is Turing equivalent ([SB96a]). The FDFD's used in this proof are those without persistent flows, without stores, and without enabling conditions that check that a flow is empty. Otherwise, we know that the problem of satisfiability in the predicate calculus is unsolvable ([LP81], p. 435). This poses the question whether these non–atomic components are fundamental to FDFD's, in that they might rise the computational power beyond that of Turing Machines, or whether they are simply

[3] Wahls has developed an interpreter for an expressive subset of FOPC assertions over abstract types. The latest reference on this work is [WBL93]. This interpreter can be viewed as an executable semantics for what he refers to as *constructive assertions*. It can also be viewed as a prototype for a CASE tool providing direct execution of abstract model–based specifications.

syntactic features included for expressive convenience. We show in this paper that FDFD's that use these features can be transformed into equivalent PFF–RDFD's that do not contain these features.

Section 3.2 provides an introduction to DFD's, FDFD's, Reduced FDFD's (RDFD's), and persistent flow–free RDFD's (PFF–RDFD's). In Section 3.3 we show that FDFD's with persistent flows, with stores, and with enabling conditions that check for empty flows can all be expressed as equivalent PFF–RDFD's without any of these features. Section 3.4 deals with the nature of the domains of values that can serve as types for values referenced in FDFD's. In the concluding Section 3.5 we stress that our goal is not to eliminate use of these features from FDFD's in actual development environments — they are expressively quite convenient — but to, without loss of generality, use PFF–RDFD's in further analysis of the computational behavior of FDFD's.

3.2 Formalized Data Flow Diagrams

Traditional DFD's are composed from four basic components: *bubbles*, *flows*, *stores*, and *terminators*. Bubbles represent either processes or procedures and are depicted as circles. In a more abstract sense, bubbles are viewed as data transformers.

Flows represent the paths over which data may travel (i. e., the roads, not the cars, in the roadmap analogy). They can represent data flow from terminators to bubbles, bubbles to bubbles, bubbles to stores, stores to bubbles, or bubbles to terminators. If we view bubbles, stores and terminators as nodes, then a DFD is simply a directed graph in which the flows are the arcs. From the perspective of a bubble, flows into the bubble are called *inflows* and flows out of the bubble are called *outflows*.

Terminators are the sources of input to and destinations of output from the system being specified. They are depicted as rectangles and can be viewed as processes that are not part of the system being specified.

In traditional DFD's, stores are viewed as data at rest (whatever that means). Stores are often just thinly veiled abstractions for files. They are often depicted as rectangles with open sides.

3.2.1 The Syntax of FDFD's

Since [Col91] provides a formal syntax and [WBL93] provides a semantics for FDFD's, we refer to the language for expressing FDFD's as DFD–SPECS. The presentation in the rest of this section is less formal than in the cited references and we continue to refer to FDFD's (rather than DFD–SPECS).

An FDFD consists of a directed graph and an associated textual part. As in traditional DFD's, the nodes of the graph are the bubbles, and the arcs are the flows.

Stores in traditional DFD's are represented with one or more inflows, representing the flow of data values to be stored, and one or more outflows, representing the flow of data values to be retrieved. This has always seemed a rather informal view of persistent repositories of data. Even if stores are just modelled as abstract data types, then a bubble adding a data value to a store would need to designate which constructor operation of the ADT is to be used. Similarly, a bubble obtaining a data value from a store would need to designate a particular selector operation.

Coleman suggests modelling stores as persistent flows with multiple originating bubbles, representing bubbles adding data values to a store, and multiple destination bubbles, representing bubbles obtaining data values from a store [Col91]. This perspective on stores as flows with multiple origin and destination bubbles is adopted in the syntax of FDFD's.

Terminators are depicted simply as bubbles in which either they have no inflows, representing the data sources, or in which they have no outflows, representing the data sinks. These bubbles are at most only partially specified.

Each bubble in the directed graph portion of an instance of an FDFD has a unique name label. Each flow is labelled with its name and type. Dashed arcs are used for *persistent* flows, while solid arcs are used for *consumable* flows. (Persistent and consumable flows are described in Subsection 3.2.2.)

The textual part of an FDFD consists of the data dictionary of types and bubble firing rules. At this specification level, these types are viewed as abstract types. They are specified using a formal, model-based approach, similar to that of [GHG+93] and [Jon86]. The firing rules defining the behavior of bubbles are expressed as assertions over the abstract types. The language for writing assertions is described in the following paragraphs and extended BNF grammar[4].

textual-part ::= [*data-dictionary*] *process**

The data dictionary defines the abstract types and abstract functions over these types used in the firing rules. The notation *type-expr*$^{*|}$ refers to union types, i. e., `type intOrReal = int | real;`.

data-dictionary ::= `Data Dictionary :` *type-decl*$^{*;}$[;] *abstract-function*$^{*;}$[;]

type-decl ::= `type` *var-name* = *type-expr*

type-expr ::= `int` | `real` | `bool` | `string` | `signal` | `set of` *type-expr* | *type-expr*$^{*|}$

`secuence of` *type-expr* | `tuple of (`*param-decl**,`)` | *type-name*

Abstract functions just serve to help modularize the specifications. An abstract function that defines type bool is really just a predicate. However, the model-based specification language does support the definition of abstract functions that define other abstract types.

[4] In this grammar, optional parts are enclosed in square brackets [ ], and the notation *expr*$^{*;}$ means a ; separated list of zero or more *expr*s.

abstract-function ::=**define** *absfun-name*(*param-decl**,) **as** *type-expr*

such that *FOPC-expr*

param-decl ::= *var-name* : *type-expr*

Each bubble is described by its name, initial state, and set of firing rules. The initial state specifies the initial values on the bubble's outflows. For flows with multiple source bubbles, the values specified must be consistent with respect to type. Each firing rule contains an enabling condition and post-condition, as discussed previously. In the enabling condition and initial state, the assertion $^{+}$*flow-name* is true exactly when at least one value is present on flow *flow-name*, while $^{-}$*flow-name* is true when no value is present. An omitted pre-condition is equivalent to just **true**.

process ::= **Process** *bubble-name* : [*initial-state*] *rule*$^{*;}$[;]

initial-state ::= **initially** *flow-enabled-list* [$\wedge$ *FOPC-expr*]

rule ::= **enabled when** *enabling-condition* **ensures** *post-condition*

enabling-condition ::= **true** | *flow-enabled-list* [$\wedge$ *FOPC-expr*]

flow-enabled-list ::= [*flow-enabled-list* $\wedge$] *flow-enabled*

flow-enabled ::= $^{+}$*flow-name* | $^{-}$*flow-name*

post-condition ::= *FOPC-expr*

The First Order Predicate Calculus used in FDFS's is augmented with operations on the built-in types, e. g., set. Unprimed flow names refer to the values on inflows, while primed flow names (') refer to outflow values. For each field of a tuple, FDFD's provide a function with the same name to extract that field from the tuple. The symbol - is used for both arithmetic subtraction and set difference, and || denotes concatenation of sequences. The **index** function provides array-like indexing into sequences, **header** returns all of its argument sequence except the last element, and **trailer** returns all of its argument sequence except the first element.

FOPC-expr ::= **true** | **false** | **not** *FOPC-expr* | *FOPC-expr* $\wedge$ *FOPC-expr* |

FOPC-expr $\vee$ *FOPC-expr* | *FOPC-expr* $\Rightarrow$ *FOPC-expr* |

$\forall$ *var-name* : *type-expr* [*FOPC-expr*] |

$\exists$ *var-name* : *type-expr* [*FOPC-expr*] |

{*FOPC-expr* | *FOPC-expr*} | (*FOPC-expr*) : *type-expr* |

int-literal | *real-literal* | *string-literal* | *bool-literal* | *var-name* |

flow-name | *flow-name'* | *absfun-name*(*FOPC-expr**,) |

unary-op(*FOPC-expr*) | *FOPC-expr binary-op FOPC-expr* |

{*FOPC-expr**,} | <*FOPC-expr**,> | (*FOPC-expr**,) |

index(*FOPC-expr*, *FOPC-expr*)

unary-op ::= *field-name* | **size** | **first** | **header** | **last** | **trailer** | **length**

binary-op ::= + | − | * | / | % | ∪ | ∩ | ∥ | = | < | ≤ | > | ≥ | ∈ | ⊂ | ⊆ | ⊃ | ⊇

3.2.2 An Informal Semantics of FDFD's

This informal description of FDFD's semantics is based on the previously referenced works ([CB94] and [LWBL96]) and on the interpreter developed by Wahls. The key concept in providing a meaning of FDFD's that allow them to serve as formal functional specifications is that of *firing* a bubble. Succinctly, firing is the process by which a bubble reads its values from its inflows and produces values on its outflows.

Bubbles fire in two steps. In the first step, a bubble reads values from its inflows, and in the second step, it writes values to its outflows. We say a bubble is *working* when it has read its inflows, but not yet produced values on its outflows. A bubble is *idle* otherwise. We treat the transitions between these states as atomic.

The effect of reading values from a flow depends on the type of the flow. When a bubble reads from a consumable flow, the value read is removed from the flow. Thus, consumable flows can be viewed as first–in, first–out unbounded queues of values, where each value is of the type associated with the flow. Reading the value of a persistent flow does not affect the flow value. When a bubble "outputs" a value to a consumable flow, that value is just appended to the back of the queue of values. Writing to a persistent flow overwrites any previous value. Thus, a persistent flow can be viewed as a variable shared between a process that writes the variable and a process that reads the variable.

Bubble firing occurs as follows. Initially, all bubbles are idle. The flows may have initial values that are specified as part of the initial state.

(i) Find the set of bubbles that may fire. This includes all bubbles in the working state, and any bubble in the idle state that has values on its inflows satisfying the enabling condition of at least one of its firing rules.

(ii) Choose one of these bubbles to fire.

(iii) Fire the bubble:

- If the bubble is *idle*:

(a) Choose one of the bubble's rules whose enabling condition is satisfied by the inflow values.

(b) Read the values referenced by this rule from the inflows. For consumable flows, remove the value. Otherwise, do not change the flow.

(c) Change the state of the bubble from *idle* to *working*.

- If the bubble is *working*:

(a) Produce values onto the outflows. These values are defined by the post–condition of the rule chosen when the bubble changed to the working state. For consumable flows, the value is enqueued. For persistent flows, the new value overwrites the flow's contents.

(b) Change the state of the bubble from *working* to *idle*.

(iv) Repeat the above steps until the set of bubbles allowed to fire in step one is empty.

3.2.3 Restricted Classes of FDFD's

As mentioned earlier, the authors have shown that a particular restricted class of FDFD's is Turing equivalent [SB96a]. In this section we define this class of FDFD's.

The first restriction is on the nature of enabling conditions. The enabling condition must do more than test for the presence of a value on a flow. For each such presence test, it must also contain an assertion limiting the value on that flow.

Definition (3.2.3.1): An enabling condition with no tests for the absence of a value on a flow (i. e., no boolean expressions ^{-}f) and in which every boolean expression ^{+}f has associated with it an assertion further bounding the value of f to a single value is called a *normal form enabling condition.* If f is a persistent flow, the ^{+}f can be omitted and only the assertion bounding the value of f is required. ■

The next restriction applies to entire FDFD's. It limits the domains of abstract types and requires normal form enabling conditions.

Definition (3.2.3.2): A *Reduced Formalized Data Flow Digram* (RDFD) is an FDFD in which (i) every abstract type modelled and referenced in the FDFD has a finite domain, (ii) sequences and tuples are restricted to a finite maximum length and (iii) every enabling condition is a normal form enabling condition. ■

Because of the finite domain and the length restriction of sequences and tuples every assertion in predicate calculus becomes solvable since the two quantifiers $\forall$ and $\exists$ are bound to a finite number of objects available. We want to stress again that, even though we do not allow sequences of arbitrary (or infinite) length as a single object, we do not prevent the production of infinite many objects of a fixed length on any of the flows.

Finally, we restrict RDFD's to preclude persistent flows:

Definition (3.2.3.3): An RDFD that does not have any persistent flows is called a *persistent flow-free Reduced (Formalized) Data Flow Diagram* (PFF-RDFD). ■

It is PFF-RDFD's that is shown to be Turing equivalent in [SB96a]. In the following three sections we argue that these restrictions can be made without loss of generality.

3.3 Transformation of FDFD's with Non-Atomic Components into PFF-RDFD's

In this section we show how to replace the test for empty flows (Example 3.3.1), persistent flows (Example 3.3.2), and stores (Example 3.3.3) by features provided by PFF-RDFD's. These examples could be easily extended to more complex situations. Obviously, another feature of FDFD's, i. e., infinite domains for flow values, can be resolved, for example, by using the unary or binary representation of objects. This type of encoding is called a *Gödel numbering*, after the logician Kurt Gödel. Thus, any object can be represented as a finite sequence of 0's and 1's.

Example (3.3.1): This example contains two bubbles. One of them (P) can always fire, producing 1's on its outflow f. The other bubble (C) will produce the value 0 on its outflow *out* as long as its inflow f is empty and the value 1 if it is not empty. The output possible on flow *out* is $\{0,1\}^*$.

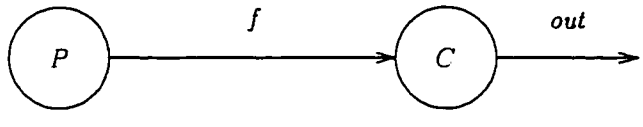

Figure 3.1: Example with Test for Empty Flow.

The assertions for the FDFD shown in Figure 3.1 using the test for empty flows (^{-}f) are specified as follows:

Process P:

$true$:

$\models f' = 1$

Process C:

^{-}f :

$\models out' = 0$:

$^{+}f \wedge f = 1$:

$\models out' = 1$

Initial State:

$^{-}f \wedge ^{-} out$

To replace the ^{-}f in the enabling condition of bubble C, we use a controller bubble Z in our PFF–RDFD (see Figure 3.2) to determine whether flow f is empty or not. When bubble P produces a value on f, it also produces a signal *fproduced* on *Pdone* to inform Z on the existence of a new value on f. However, the value on f will not immediately be available for C since Z continues to send *fisempty* on flow *Cenab* as long as *fcount* $= 0$ holds. Instead, Z has to consume the signal on *Pdone* first and increment the counter *fcount*. Then, since *fcount* ≥ 1 holds, the next value on *Cenab* will be *fnotempty*, allowing C to consume the value on f. The output possible on flow *out* is $\{0,1\}^*$ as in the original example.

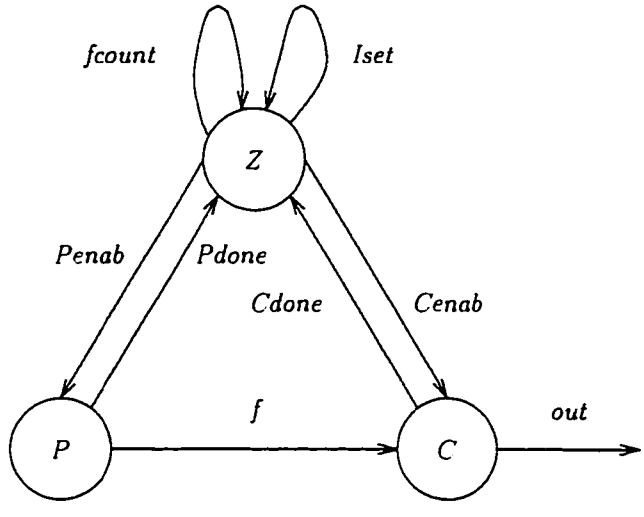

Figure 3.2: Example without Test for Empty Flow.

For the ease of our description we informally use the unbounded integer variable *fcount*. Using the unary or binary representation of this variable would solve the problem of an infinite number of integer objects but it would also extend the coding of this example which is not desired.

Initially, *fcount* contains 0, i. e., no value is available on flow *f*. Flow *Iset* indicates which bubbles are idle, i. e., initially *P* and *C*. Possible values on the flows are *PC*, *P*, *C*, and *O* (which indicates neither *P* nor *C* is idle) on *Iset*, *fisempty* and *fnotempty* on *Cenab*, *fconsumed* and *nofconsumed* on *Cdone*, *go* on *Penab*, and *fproduced* on *Pdone*.

Process Z:

$^{+}Iset \wedge Iset = PC \wedge^{+} fcount \wedge fcount = 0$:

$\models Penab' = go \wedge Iset' = C \wedge fcount' = fcount$

$\Box Cenab' = fisempty \wedge Iset' = P \wedge fcount' = fcount$;

$^{+}Iset \wedge Iset = PC \wedge^{+} fcount \wedge fcount \geq 1$:

$\models Penab' = go \wedge Iset' = C \wedge fcount' = fcount$

$\Box Cenab' = fnotempty \wedge Iset' = P \wedge fcount' = fcount$;

$^{+}Iset \wedge Iset = P \wedge^{+} fcount \wedge fcount \geq 0$:

$\models Penab' = go \wedge Iset' = O \wedge fcount' = fcount$;

$^{+}Iset \wedge Iset = C \wedge^{+} fcount \wedge fcount = 0$:

$\models Cenab' = fisempty \wedge Iset' = O \wedge fcount' = fcount$;

$^{+}Iset \wedge Iset = C \wedge^{+} fcount \wedge fcount \geq 1$:

$\models Cenab' = fnotempty \wedge Iset' = O \wedge fcount' = fcount$;

$^{+}Pdone \wedge Pdone = fproduced \wedge^{+} Iset \wedge Iset = O \wedge^{+} fcount \wedge fcount \geq 0$:

$\models Iset' = P \wedge fcount' = fcount + 1$;

$^{+}Pdone \wedge Pdone = fproduced \wedge^{+} Iset \wedge Iset = C \wedge^{+} fcount \wedge fcount \geq 0$:

$\models Iset' = PC \wedge fcount' = fcount + 1$;

$^{+}Cdone \wedge Cdone = nofconsumed \wedge^{+} Iset \wedge Iset = O \wedge^{+} fcount \wedge fcount \geq 0$:

$\models Iset' = C \wedge fcount' = fcount$;

$^{+}Cdone \wedge Cdone = nofconsumed \wedge^{+} Iset \wedge Iset = P \wedge^{+} fcount \wedge fcount \geq 0$:

$\models Iset' = PC \wedge fcount' = fcount$;

$^{+}Cdone \wedge Cdone = fconsumed \wedge^{+} Iset \wedge Iset = O \wedge^{+} fcount \wedge fcount \geq 1$:

$\models Iset' = C \wedge fcount' = fcount - 1$;

$^{+}Cdone \wedge Cdone = fconsumed \wedge^{+} Iset \wedge Iset = P \wedge^{+} fcount \wedge fcount \geq 1$:

$\models Iset' = PC \wedge fcount' = fcount - 1$

Process P:

$^{+}Penab \wedge Penab = go$:

$\models f' = 1 \wedge Pdone' = fproduced$

Process C:

$^{+}Cenab \wedge Cenab = fisempty$:

$\models out' = 0 \wedge Cdone' = nofconsumed$;

$^{+}Cenab \wedge Cenab = fnotempty \wedge^{+} f \wedge f = 1$:

$\models out' = 1 \wedge Cdone' = fconsumed$

Initial State:

$fcount = 0 \wedge Iset = PC \wedge^{-} f \wedge^{-} out \wedge^{-} Penab \wedge^{-} Pdone \wedge^{-} Cenab \wedge^{-} Cdone$ ■

Example (3.3.2): As in the previous example, this example also contains two bubbles. Here bubble A can always fire, producing 0's and 1's on its persistent outflow f. Bubble B will produce the value 0 on its outflow *out* if it reads a 0 on its inflow f and the value 1 if it reads a 1. The output possible on flow *out* is $\{0,1\}^{*}$.

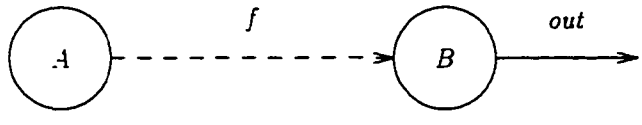

Figure 3.3: Example with Persistent Flow.

The assertions for the FDFD shown in Figure 3.3 are specified as follows:

Process A:

true :

$\models f' = 0$

$\Box f' = 1$

Process B:

$f = 0$:

$\models out' = 0$:

$f = 1$:

$\models out' = 1$

Initial State:

$(f = 0 \vee f = 1) \wedge^{-} out$

This time, we use the controller bubble Z in our PFF–RDFD (see Figure 3.4) to inform bubble B on the latest value that has been generated by bubble A. However, instead of writing this new value on a persistent flow, A forwards this new value to Z. Once Z has read this value from its inflow *Adone*, it will be stored on flow *fval*. The next value on flow *Benab* will be the value currently stored in *fval*. The output possible on flow *out* is $\{0,1\}^*$ as in the original example.

We are using the somewhat sloppy notation 0/1 where we mean that either the value 0 or the value 1 is available. It should be obvious how to extend this notation such that the mappings are in accordance with our definition of a PFF–RDFD.

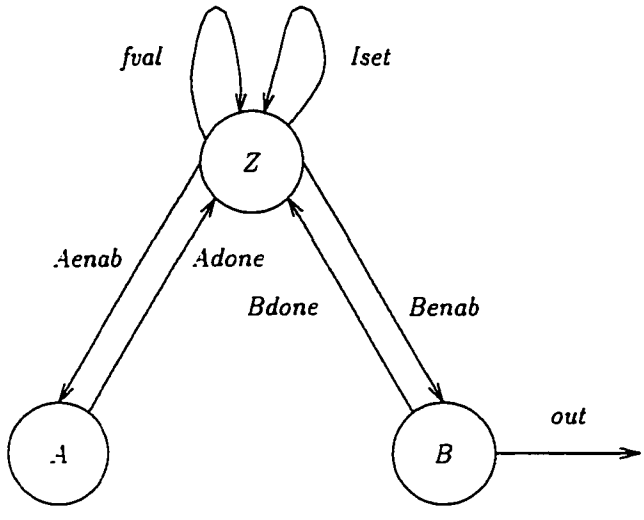

Figure 3.4: Example without Persistent Flow.

Initially, *fval* contains either a 0 or a 1. Flow *Iset* indicates which bubbles are idle, i. e., initially A and B. Possible values on the flows are AB, A, B, and O (which indicates neither A nor B is idle) on *Iset*, 0 and 1 on *Benab*, *done* on *Bdone*, *go* on *Aenab*, and 0 and 1 on *Adone*.

Process Z:

$^{+}Iset \wedge Iset = AB \wedge^{+} fval \wedge fval = 0/1$:

$\models Aenab' = go \wedge Iset' = B \wedge fval' = fval$

$\Box Benab' = fval \wedge Iset' = A \wedge fval' = fval$;

$^{+}Iset \wedge Iset = A \wedge^{+} fval \wedge fval = 0/1$:

$\models Aenab' = go \wedge Iset' = O \wedge fval' = fval;$

$^{+}Iset \wedge Iset = B \wedge^{+} fval \wedge fval = 0/1 :$

$\models Benab' = fval \wedge Iset' = O \wedge fval' = fval;$

$^{+}Adone \wedge Adone = 0 \wedge^{+} Iset \wedge Iset = O \wedge^{+} fval \wedge fval = 0/1 :$

$\models Iset' = A \wedge fval' = 0;$

$^{+}Adone \wedge Adone = 1 \wedge^{+} Iset \wedge Iset = O \wedge^{+} fval \wedge fval = 0/1 :$

$\models Iset' = A \wedge fval' = 1;$

$^{+}Adone \wedge Adone = 0 \wedge^{+} Iset \wedge Iset = B \wedge^{+} fval \wedge fval = 0/1 :$

$\models Iset' = AB \wedge fval' = 0;$

$^{+}Adone \wedge Adone = 1 \wedge^{+} Iset \wedge Iset = B \wedge^{+} fval \wedge fval = 0/1 :$

$\models Iset' = AB \wedge fval' = 1;$

$^{+}Bdone \wedge Bdone = done \wedge^{+} Iset \wedge Iset = O \wedge^{+} fval \wedge fval = 0/1 :$

$\models Iset' = B \wedge fval' = fval;$

$^{+}Bdone \wedge Bdone = done \wedge^{+} Iset \wedge Iset = A \wedge^{+} fval \wedge fval = 0/1 :$

$\models Iset' = AB \wedge fval' = fval$

Process A:

$^{+}Aenab \wedge Aenab = go :$

$\models Adone' = 0$

$\square Adone' = 1$

Process B:

$^{+}Benab \wedge Benab = 0 :$

$\models out' = 0 \wedge Bdone' = done;$

$^{+}Benab \wedge Benab = 1 :$

$\models out' = 1 \wedge Bdone' = done$

Initial State:

$(fval = 0 \vee fval = 1) \wedge Iset = AB \wedge^{-} out \wedge^{-} Aenab \wedge^{-} Adone \wedge^{-} Benab \wedge^{-} Bdone$ ■

Stores are a common feature of traditional DFD's. They are usually used to represent persistent data, often with the intended implementation using files. Thus stores are usually represented with inflows representing data to be added to the store and outflows with data retrieved from the store. Since there has been no formalism for representing different "constructor" operations to add to a store and no formalism for representing different "selector" or "query" operations for getting data from a

store, stores have not been included in FDFD's in [LWBL96], although a possible extension has been mentioned.

However, the question of whether stores, i. e., the usual way stores are used in traditional DFD's, can be modeled using just the features of PFF–RDFD's is pertinent.

Example (3.3.3): This example demonstrates how to replace the common notion of stores as used in DFD's. Here, we have two bubbles A and B that both read from and write to the same store *Store*. Note that, due to the two phase firing semantics of FDFD's, the value of a flow is read in step 1 while a new value of a flow is written in step 2. For example, when A issues a write command (the value 0), any number of reads from B may return the old value. Even a write from B (the value 1) started later than A's write may be completed earlier than A's write, leaving the value 0 as the final value.

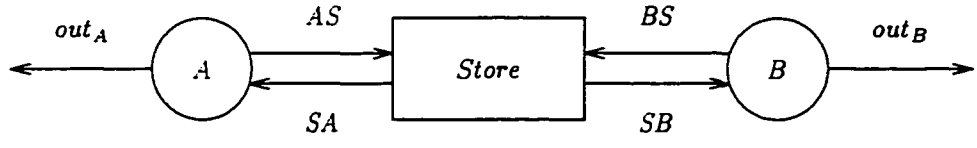

Figure 3.5: Example with Store.

The assertions for the FDFD shown in Figure 3.5 are specified as follows:

Process A:

^{-}SA :

$\models AS' = 0$;

$SA = 0$:

$\models AS' = 0$

$\Box out'_A = 0$;

$SA = 1$:

$\models AS' = 0$

$\Box out'_A = 1$

Process B:

^{-}SB :

$\models BS' = 1$;

$SB = 0$:

$\models BS' = 1$

$\Box out'_B = 0$;

$SB = 1$:

$\models BS' = 1$

$\Box out'_B = 1$

Initial State:

$^{-}SA \wedge^{-} SB \wedge^{-} AS \wedge^{-} BS \wedge^{-} out_A \wedge^{-} out_B$

Once again, we make use of a controller bubble Z in our PFF-RDFD (see Figure 3.6). This time, it is used to guarantee that a write does not modify the stored value before all reads issued prior to or during the write have finished returning the old value. If one bubble issues a write while the other bubble's write has not yet finished, the order in which their values are stored is determined nondeterministically.

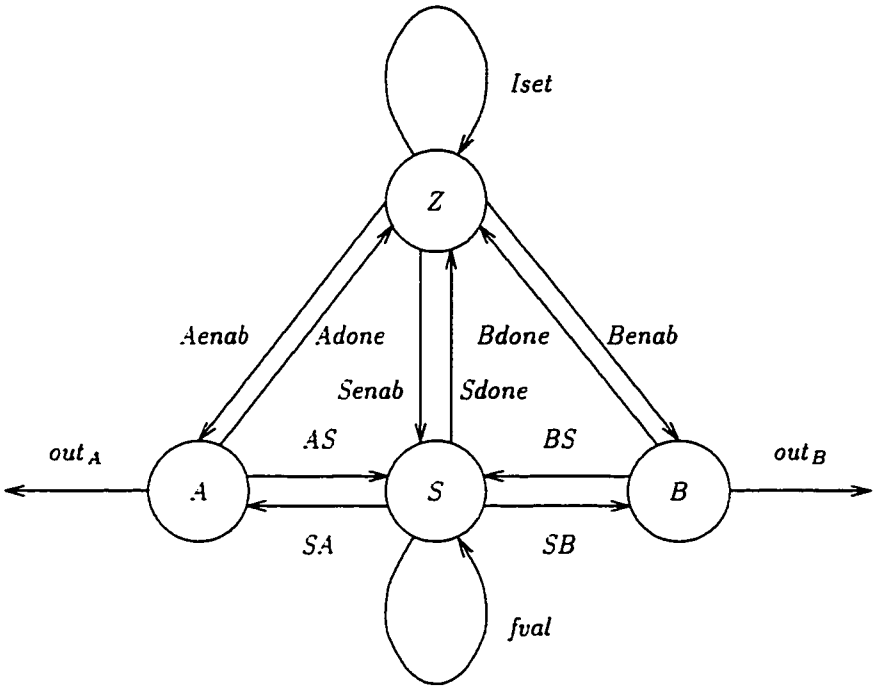

Figure 3.6: Example without Store.

Initially, *fval* contains either a 0 or a 1. Flow *Iset* indicates which bubbles are idle, i. e., initially A, B, and S. Possible values on the flows are ABS, AS, BS, A, B, S, and O (which indicates neither A nor B nor S is idle) on *Iset*, *request* and *write0* on AS, *request* and *write1* on BS, 0 and 1 on SA, on SB, and on *fval*, *go* on *Aenab* and on *Benab*, *fread* and *fproduced* on *Adone* and on *Bdone*, *Aproduced* and *Bproduced* on *Senab*, and *Awritten* and *Bwritten* on *Sdone*.

Process Z:

$^{+}Iset \wedge Iset = ABS$:

$\models Aenab' = go \wedge Iset' = BS$

$\Box Benab' = go \wedge Iset' = AS$;

$^{+}Iset \wedge Iset = AS$:

$\models Aenab' = go \wedge Iset' = S$;

$^{+}Iset \wedge Iset = BS$:

$\models Benab' = go \wedge Iset' = S$;

$^{+}Adone \wedge Adone = fread \wedge^{+} Iset \wedge Iset = S$:

$\models Iset' = AS$;

$^{+}Adone \wedge Adone = fread \wedge^{+} Iset \wedge Iset = BS$:

$\models Iset' = ABS$;

$^{+}Adone \wedge Adone = fproduced \wedge^{+} Iset \wedge Iset = S$:

$\models Senab' = Aproduced \wedge Iset' = O$;

$^{+}Adone \wedge Adone = fproduced \wedge^{+} Iset \wedge Iset = BS$:

$\models Senab' = Aproduced \wedge Iset' = B$;

$^{+}Bdone \wedge Bdone = fread \wedge^{+} Iset \wedge Iset = S$:

$\models Iset' = BS$;

$^{+}Bdone \wedge Bdone = fread \wedge^{+} Iset \wedge Iset = AS$:

$\models Iset' = ABS$;

$^{+}Bdone \wedge Bdone = fproduced \wedge^{+} Iset \wedge Iset = S$:

$\models Senab' = Bproduced \wedge Iset' = O$;

$^{+}Bdone \wedge Bdone = fproduced \wedge^{+} Iset \wedge Iset = AS$:

$\models Senab' = Bproduced \wedge Iset' = A$;

$^{+}Sdone \wedge Sdone = Awritten \wedge^{+} Iset \wedge Iset = O$:

$\models Iset' = AS$;

$^{+}Sdone \wedge Sdone = Awritten \wedge^{+} Iset \wedge Iset = B$:

$\models Iset' = ABS$;

$^{+}Sdone \wedge Sdone = Bwritten \wedge^{+} Iset \wedge Iset = O$:

$\models Iset' = BS$;

$^{+}Sdone \wedge Sdone = Bwritten \wedge^{+} Iset \wedge Iset = A$:

$\models Iset' = ABS$

Process A:

$^{+}Aenab \wedge Aenab = go$:

$\models AS' = request$

$\Box AS' = write0 \wedge Adone' = fproduced$;

$^{+}SA \wedge SA = 0/1$:

$\models out'_A = SA \wedge Adone' = fread$

Process B:

$^{+}Benab \wedge Benab = go$:

$\models BS' = request$

$\Box BS' = write1 \wedge Bdone' = fproduced$;

$^{+}SB \wedge SB = 0/1$:

$\models out'_B = SB \wedge Bdone' = fread$

Process S:

$^{+}AS \wedge AS = request \wedge^{+} fval \wedge fval = 0/1$:

$\models SA' = fval \wedge fval' = fval$;

$^{+}AS \wedge AS = write0 \wedge^{+} fval \wedge fval = 0/1 \wedge Senab = Aproduced$:

$\models fval' = 0 \wedge Sdone' = Awritten$;

$^{+}BS \wedge BS = request \wedge^{+} fval \wedge fval = 0/1$:

$\models SB' = fval \wedge fval' = fval$;

$^{+}BS \wedge BS = write1 \wedge^{+} fval \wedge fval = 0/1 \wedge Senab = Bproduced$:

$\models fval' = 1 \wedge Sdone' = Bwritten$

Initial State:

$(fval = 0 \vee fval = 1) \wedge Iset = ABS \wedge^{-} out_A \wedge^{-} out_B$

$\wedge^{-} Aenab \wedge^{-} Adone \wedge^{-} Benab \wedge^{-} Bdone \wedge^{-} Senab \wedge^{-} Sdone \wedge^{-} SA \wedge^{-} SB \wedge^{-} AS \wedge^{-} BS$ ■

3.4 Transformation of FDFD's with Infinite Domains into PFF–RDFD's

While in the previous section we have considered non-atomic components of FDFD's that are of particular interest for the Structured Analysis, we will now deal with a more theoretical issue, that is, infinite domains. We have mentioned several times that we have to restrict every abstract type to a finite domain and do not allow sequences, tupels, and sets of arbitrary (or infinite) length. The reasons for this restriction follow in the next four subsections.

3.4.1 Quantifiers over Unbounded Sets

Consider the case where the enabling condition reads as

$$^{+}f \wedge \exists x_1 : \mathtt{int} \ldots \exists x_n : \mathtt{int}\,(P(x_1, \ldots, x_n, f) = 0) \models C_{true}$$

where flow f is of type `int`, and P is a given polynomial of degree $n+1$. The above condition represents a diophantine equation. The problem, whether there exists a procedure which in a finite number of steps enables one to determine whether or not a given diophantine equation has an integer solution is known as Hilbert's 10th problem. It has been shown ([Mat70]) that this problem is undecidable. Therefore, we can express something as a condition for an FDFD, but we are not capable to provide an algorithm that allows us to decide whether this condition holds or does not hold.

However, assume we define `bounded_int = {minint, ..., maxint}` and redefine the enabling condition as

$$^{+}f \wedge \exists x_1 : \mathtt{bounded\_int} \ldots \exists x_n : \mathtt{bounded\_int}\,(P(x_1, \ldots, x_n, f) = 0) \models C_{true}$$

where flow f is of type `bounded_int`, too. Now this condition becomes decidable for any possible value on flow f since there are only finite many cases that have to be considered. Even more, we could reduce this condition to our normal form enabling condition when designing the model, i. e.,

$$^{+}f \wedge f = \mathtt{minint} \models C_{true}$$
$$^{+}f \wedge f = \mathtt{minint} + 1 \models C_{true}$$
$$^{+}f \wedge f = \mathtt{minint} + 2 \models C_{true}$$
$$\vdots$$
$$^{+}f \wedge f = \mathtt{maxint} \models C_{true}$$

where only those cases are listed that result in *true* in the original condition.

3.4.2 Infinite Domains

Assume we have a flow f of type `unsigned int` and only distinguish among the values $0, \ldots, n$:

$$^{+}f \wedge f = 0 \models C_0$$
$$^{+}f \wedge f = 1 \models C_1$$
$$\vdots$$
$$^{+}f \wedge f = n \models C_n$$
$$^{+}f \wedge f > n \models C_{>n}$$

As we have mentioned several times, we can use the unary representation of (unsigned) integers, where the sequence of values $(\underbrace{1, \ldots, 1}_{n}, 0)$ represents n. With the help of an additional flow *last* (initialized

with "0") , we can redesign the previous conditions:

$$^{+}f \wedge f = 0 \wedge^{+} last \wedge last = \text{"0"} \models C_0$$

$$^{+}f \wedge f = 1 \wedge^{+} last \wedge last = \text{"0"} \models last' = \text{"1"}$$

$$^{+}f \wedge f = 0 \wedge^{+} last \wedge last = \text{"1"} \models C_1$$

$$^{+}f \wedge f = 1 \wedge^{+} last \wedge last = \text{"1"} \models last' = \text{"2"}$$

$$\vdots$$

$$^{+}f \wedge f = 0 \wedge^{+} last \wedge last = \text{"n"} \models C_n$$

$$^{+}f \wedge f = 1 \wedge^{+} last \wedge last = \text{"n"} \models last' = \text{">n"}$$

$$^{+}f \wedge f = 0 \wedge^{+} last \wedge last = \text{">n"} \models C_{>n}$$

$$^{+}f \wedge f = 1 \wedge^{+} last \wedge last = \text{">n"} \models last' = \text{">n"}$$

Now the domain, i. e., the objects that are used on all flows, is finite, i. e., the set $\{0, 1\} \cup \{$"0", "1",..., "n", ">n"$\}$.

The idea to use the unary or binary representation of objects is used throughout the theoretical literature in computer science and this type of encoding is called a *Gödel numbering*, after the logician Kurt Gödel. Of course, we can use the Gödel numbering not only to distinguish among different cases as seen above, but also to do calculus directly based on this representation.

3.4.3 Unbounded Sequences, Types, and Sets

Assume the length of a sequence can grow beyond any bound during the execution of the FDFD, however it remains finite at any time. Instead of sending a single object of type **secuence**, we can remodel our FDFD such that each element of the sequence is written as a single object to the unbounded FIFO flow and an additional delimiter is used to indicate the end of the sequence. The same mechanism can be used for types and sets.

3.4.4 Infinite Sequences, Types, and Sets

Assume a flow f of type `set of int` contains a single object `int`, i. e., the infinite many values of the set integer. Consider a bubble that reads its input from flow f and produces some output on flow *out* is specified as follows:

$$^{+}f \wedge f = \texttt{int} \models out' = \{i + 1 \mid i \in f\}$$

Hence, this bubble is intended to produce the successor of each of the values of its input set. What is the semantics underlying the above construct? In an abstract world, we may assume that we can yield the successors of an infinte set in finite time. However, it is more realistic to assume that we need finite

time (> 0) to obtain the successor of each element. Thus, we need infinite time to yield the successors of an infinite set. Therefore, we should assume that this transition once enabled never terminates.

This behavior to spend infinite time on determing the successors can be remodeled the following way: One bubble produces the infinite set `int` on flow *f* using the unary representation introduced in Subsection 3.4.2. Another bubble consumes the head element from flow *f* and produces the output on flow *out*. Both bubbles will alternate between *idle* and *working*, but none of them is expected to reach a state where it finally remains *idle*. Thus, there exists an infinite firing sequence that only uses these two bubbles. Also, another bubble *b* that makes use of flow *out* as its input can be remodeled to proceed only if it reads an additional delimiter on *out*. However, this delimiter will never be send, so *b* makes no overall progress (except reading from *out* and eventually storing these values on a flow with *b* as source and destination). The same mechanism can be used for sequences and types.

3.5 Summary

In this paper, we have shown how to remodel FDFD's into PFF–RDFD's. We have given examples where the test for empty flows (Example 3.3.1), persistent flows (Example 3.3.2), and stores (Example 3.3.3) have been replaced by features provided by PFF–RDFD's. We have also shown why the use of another feature of FDFD's, i. e., infinite domains, has to be restricted, and how to do so using the unary or binary representation of objects.

Even though we have not provided a general algorithm that transfers any given FDFD into a PFF–RDFD, it should be obvious how our idea could be easily extended to more complex situations. In particular, we know that PFF–RDFD's have the computational power of Turing Machines ([SB96a]). From the given examples, it should be obvious that any additional non–atomic component only adds to the expressive power of the model but does not allow to model features unavailable for the basic PFF–RDFD and Turing Machine, respectively.

Acknowledgements

Symanzik's research was partially supported by a German "DAAD–Doktorandenstipendium aus Mitteln des zweiten Hochschulsonderprogramms".

4 SUBCLASSES OF FORMALIZED DATA FLOW DIAGRAMS: MONOGENEOUS, LINEAR, AND TOPOLOGICALLY FREE CHOICE RDFD'S

A paper submitted to Acta Informatica

Jürgen Symanzik[1] and Albert L. Baker[2]

Abstract

Formalized Data Flow Diagrams (FDFD's) and, especially, Reduced Data Flow Diagrams (RDFD's) are Turing equivalent ([SB96a]). Therefore, no decidability problem can be solved for FDFD's in general. However, it is possible to define subclasses of FDFD's for which decidability problems can be answered.

In this paper we will define certain subclasses of FDFD's, which we call Monogeneous RDFD's, Linear RDFD's, and Topologically Free Choice RDFD's. We will show that two of these three subclasses of FDFD's can be simulated via isomorphism by the correspondingly named subclasses of FIFO Petri Nets. It is known that isomorphisms between computation systems guarantee the same answers to corresponding decidability problems (e. g., reachability, deadlock, liveness) in the two systems ([KM82]). This means that problems where it is known that they can (not) be solved for a subclass of FIFO Petri Nets it follows immediately that the same problems can (not) be solved for the correspondingly named subclass of FDFD's.

Keywords

Decidability Problem, Computation System, Isomorphism, FIFO Petri Net.

[1] Primary researcher and author.
[2] Professor, Department of Computer Science, Iowa State University.

4.1 Introduction

Formalized Data Flow Diagrams (FDFD's) as given in [LWBL96] are a relatively new approach to the formalization of traditional Data Flow Diagrams (DFD's). Recently it has been formally established that FDFD's are Turing equivalent ([SB96a]) and their non-atomic components, e. g., stores and persistent flows, are not essential to the expressive power of FDFD's ([SB96b]). Unfortunately, this equivalence to Turing Machines prevents the analytical solution of decidability problems (e. g., reachability, deadlock, liveness) for FDFD's.

However, there exist subclasses of another computational model with the computational power of Turing Machines, FIFO Petri Nets (introduced in [MM81]), for which decidability problems can be solved. Many variations and restrictions of the basic model of FIFO Petri Nets have been considered, e. g., in [FM82], [Sta83], [Fin84], [FR85], [MF85], [Fin86], [Rou87], [CF87], [FC88], [FR88], and [Fan92]. Probably the most important work done with respect to this current paper was the survey on decidability questions for subclasses of FIFO Petri Nets in [FR88]. There, it was established which decidability problems can be solved for which subclasses of FIFO Petri Nets typically considered in the literature, that is, Monogeneous FIFO Petri Nets, Linear FIFO Petri Nets, and Topologically Free Choice FIFO Petri Nets.

In this paper, we first summarize required definitions and main results for computation systems, FIFO Petri Nets, and decidability problems in Section 4.2. In Section 4.3, we define subclasses of Reduced Data Flow Diagrams (RDFD's), i. e., Monogeneous RDFD's, Linear RDFD's, and Topologically Free Choice RDFD's. From [SB96a] we know that every RDFD can be simulated by a FIFO Petri Net with respect to an isomorphism h. We will show that this isomorphism h actually maps Monogeneous persistent flow-free RDFD's and Linear RDFD's onto subclasses of FIFO Petri Nets of the same names. Moreover, from [KM82] we know that isomorphisms preserve many decidability problems. Therefore, we can conclude that a problem that is decidable for a subclass of FIFO Petri Nets is also decidable for the related subclass of FDFD's. Unfortunately, our isomorphism h does not map (Extended) Topologically Free Choice RDFD's to (Extended) Topologically Free Choice FIFO Petri Nets. We finish this paper with a summary on possible future research in Section 4.4.

4.2 Definitions

In the next two subsections, we summarize definitions and results from [KM82]. Please refer to this work for a more detailed explanation of symbols and for additional definitions. A short summary of [KM82] is given in [SB96a]. We assume that the reader is familiar with [SB96a] since our notations, definitions, and proofs of theorems are closely related to this reference. In Subsections 4.2.3 and 4.2.4, we summarize definitions for FIFO Petri Nets and related decidability problems. In Subsection 4.2.5, we deal with subclasses of FIFO Petri Nets.

4.2.1 Computation Systems

Definition (4.2.1.1): A *computation system* $S = (\Sigma, D, x, \overline{\ })$ consists of a set D, an element x of D, a finite set Σ of *operations*, and a function "$\overline{\ }$" from Σ to the set of partial functions from D to D. That is, for each $a \in \Sigma$, $\overline{a}$ is a partial function from D to D. The function "$\overline{\ }$" is extended to Σ^* by $\overline{\lambda} =$ identity, $\overline{\alpha\beta}(y) = \overline{\alpha} \cdot \overline{\beta}(y) = \overline{\beta}(\overline{\alpha}(y))$, $\alpha \cdot \beta \in \Sigma^*, y \in D$. ■

4.2.2 Decidability Problems for Computation Systems

Definition (4.2.2.1): For a given computation system $S = (\Sigma, D, x, \overline{\ })$, we are interested in answers to the following decidability problems:

(i) Reachability: For $y \in D$, is $y \in R_S$?

(ii) Deadlock: Does there exist an $\alpha \in C_S$ such that, for every $a \in \Sigma$, $\alpha a \notin C_S$?

(iii) Termination: Is C_S finite?

(iv) Finiteness: Is R_S finite?

(v) Equivalence of sets of computation sequences: For $y, z \in D$, is $C_S(y) = C_S(z)$?

(vi) Liveness: For any $\alpha \in C_S$ and $a \in \Sigma$, does there exist a $\beta \in \Sigma^*$ such that $\alpha\beta a \in C_S$?

(vii) Exceedability: With D a partially ordered set[3] and given $y \in D$, does there exist a $z \in R_S$ such that $z \geq y$? ■

In this definition, R_S denotes the reachability set from x, C_S the set of all finite computation sequences from x, and $C_S(y)$ the set of all finite computation sequences from y.

[3] $(D, \geq)$ can be any partial ordering on the set D.

Definition (4.2.2.2): Let $S_1 = (\Sigma_1, D_1, x_1, \overline{\ }^1)$ and $S_2 = (\Sigma_2, D_2, x_2, \overline{\ }^2)$ be computation systems. A *homomorphism* $h = (\tau, \rho) : S_1 \to S_2$ consists of a homomorphism $\tau : \Sigma_1^* \to \Sigma_2^*$, and an injection $\rho : D_1 \to D_2$ which satisfies the following conditions:

$$\rho(x_1) = x_2 \qquad (4.2.2.2.1)$$

$$\forall y, z \in R_{S_1} \forall \alpha \in \Sigma_1^* : (y \xrightarrow{\alpha} z \Rightarrow \rho(y) \xrightarrow{\tau(\alpha)} \rho(z)) \qquad (4.2.2.2.2)$$

■

Definition (4.2.2.3): Let $h = (\tau, \rho) : S_1 \to S_2$ be a homomorphism. h is called an *isomorphism* if there is a homomorphism $h' = (\tau', \rho') : S_2 \to S_1$ such that $hh' : S_2 \to S_2$ and $h'h : S_1 \to S_1$ are identities, i. e., $\tau\tau' : \Sigma_2^* \to \Sigma_2^*$, $\tau'\tau : \Sigma_1^* \to \Sigma_1^*$, $\rho\rho' : D_2 \to D_2$, and $\rho'\rho : D_1 \to D_1$ are identities. ■

Definition (4.2.2.4): Let H be a class of homomorphisms. Let P be a problem of the form: Given a computation system $S = (\Sigma, D, x, \overline{\ })$, $y_1, \ldots, y_n \in D$, whether $P(S, y_1, \ldots, y_n)$? We say that P is *preserved* by H under the following condition: For any S_1 and S_2, if there is an $h \in H$ such that $h = (\tau, \rho) : S_1 \to S_2$, then $P(S_1, y_1, \ldots, y_n)$ holds if, and only if, $P(S_2, \rho(y_1), \ldots, \rho(y_n))$ holds. ■

Theorem (4.2.2.5): Particular types of homomorphisms preserve the following decidability problems:

(i) A spanning homomorphism preserves reachability.

(ii) A spanning homomorphism preserves deadlock.

(iii) A spanning homomorphism preserves the termination property.

(iv) A surjective homomorphism preserves finiteness.

(v) A principal homomorphism preserves equivalence of sets of computation sequences.

(vi) A principal homomorphism preserves liveness.

(vii) An order preserving spanning homomorphism preserves exceedability.

Proof: Proofs are given in [KM82], Section 4. ■

It should be noted that an isomorphism h is also a bijective (hence injective, surjective, hence spanning), length preserving, and principal homomorphism. Thus, an isomorphism h preserves decidability problems (i) to (vi).

4.2.3 FIFO Petri Nets

In some sense, FIFO Petri Nets (introduced in [MM81]) are Petri Nets (see [Pet81], for example) where places contain words instead of tokens and arcs are labelled by words. More formally, we make use of the definition of FIFO Petri Nets as given in [Rou87].

Definition (4.2.3.1): A *FIFO Petri Net* is a quintuple $FPN = (P, T, B, F, Q)$ where P is a finite set of *places* (also called *queues*), T is a finite set of *transitions* (disjoint from P), Q is a finite *queue alphabet*, and $F : T \times P \to Q^*$ and $B : P \times T \to Q^*$ are two mappings called respectively *forward* and *backward incidence mappings*. ■

Definition (4.2.3.2): A *marking* M of a FIFO Petri Net is a mapping $M : P \to Q^*$.
A transition t is *fireable* in M, written $M(t >$, if $\forall p \in P : B(p,t) \leq M(p)$ (where $u \leq x$ means u is a prefix of x).

For a marking M, we define the firing of a transition t, written $M(t > M'$, if $M(t >$ and the following equation between words holds $\forall p \in P : B(p,t)M'(p) = M(p)F(t,p)$. That means, the firing of a transition t removes $B(p,t)$ from the head of $M(p)$ and appends $F(t,p)$ to the end of the resulting word. ■

Definition (4.2.3.3): A FIFO Petri Net FPN together with an initial marking $M_0 : P \to Q^*$ is called a *marked FIFO Petri Net* and is denoted by (FPN, M_0).

As usual, the firing of a transition can be extended to the firing of a sequence of transitions. We denote by $FS(FPN, M_0)$ the *set of firing sequences* of this FIFO Petri Net. The firing of a sequence u of transitions from a marking M to a marking M' is written as $M(u > M'$.

The set of markings that are reachable from M_0 is called *reachability set* and it is denoted by $Acc(FPN, M_0)$. ■

In addition, the following two definitions from [FR88] are used within this paper:

Definition (4.2.3.4): Let $R(FPN, M_0)$ denote the set of all markings that are reachable from M_0, i. e., $R(FPN, M_0) = \{M \mid \exists x \in T^* : M_0(x > M\}$. Let $L(FPN, M_0)$ denote the language of the net or the set of all sequences in T^* that are fireable from M_0, i. e., $L(FPN, M_0) = \{x \mid x \in T^* \wedge M_0(x >\}$. An element $x \in T^*$ is said to be in the center of (FPN, M_0), denoted by $C(FPN, M_0)$, if, and only if, $M_0(x > M$ and $L(FPN, M)$ is infinite. ■

However in accordance with many other references, we prefer the abbreviations FS for the set of firing sequences (language, set of computation sequences) and RS for the reachability set. Therefore, we denote by $FS(FPN, M_0)$ what is denoted by $L(FPN, M_0)$ in [FR88] and we denote by $RS(FPN, M_0)$ what is denoted by $R(FPN, M_0)$ in [FR88] and by $Acc(FPN, M_0)$ in [Rou87].

Definition (4.2.3.5): The *input language* of a place p in (FPN, M_0) is defined as $L_I(FPN, M_0, p) = h_p(FS(FPN, M_0))$ with $h_p(t) = F(t, p)$. ■

4.2.4 Decidability Problems for FIFO Petri Nets

The following definition is due to [FR88].

Definition (4.2.4.1): For a given marked FIFO Petri Net (FPN, M_0), we define the following decidability problems:

Total Deadlock Problem (TDP): Is $FS(FPN, M_0)$ finite?

Partial Deadlock Problem (PDP): Is there a finite path in (FPN, M_0) that can not be extended, i. e., does there exist an $x \in T^*$ such that $M_0(x > M$ where no transition in T is fireable from M?

Boundedness Problem (BP): Is $RS(FPN, M_0)$ finite?

Reachability Problem (RP): For a marking M, is $M \in RS(FPN, M_0)$?

Quasi–Liveness Problem (QLP): $\forall t \in T$, is there an $x \in T^*$ such that $M_0(xt > ?$

Liveness Problem (LP): $\forall M \in RS(FPN, M_0)$ $\forall t \in T$, is there an $x \in T^*$ such that $M(xt > ?$

Center Problem (CP): Is there an algorithm that will generate a recursive representation of $C(FPN, M_0)$?

Regularity Problem (RegP): Is $FS(FPN, M_0)$ regular? ■

Unfortunately, names for decidability problems for computation systems in [KM82] and FIFO Petri Nets in [FR88] differ. Other terms can be found within the literature. It should be noted that the following names for decidabilty problems are identical:

Computation Sytsem	FIFO Petri Net
Reachability	RP
Deadlock	PDP
Termination	TDP
Finiteness	BP
Equivalence	
Liveness	LP
Exceedability	
	QLP
	CP
	RegP

4.2.5 Subclasses of FIFO Petri Nets

In this subsection, we summarize definitions and theorems related to Monogeneous FIFO Petri Nets (e. g., [Sta83], [Fin84], [MF85], [Fin86], [Rou87], [FR88]), Linear FIFO Petri Nets (e. g., [CF87], [FR88]), and and (Extended) Topologically Free Choice FIFO Petri Nets (e. g., [Fin86], [Rou87], [FC88], [FR88]).

Monogeneous FIFO Petri Nets

In this part, we follow the notation in [Fin86] and [FR88].

Definition (4.2.5.1): Let A be a finite alphabet. Let L be a language on A. Let x and y be words in L. x is called a *left factor* of y, $x \leq y$ in symbols, if $\exists$ word $z \in A^* : xz = y$.

For a language $L \subset A^*$, we denote by $LeftFactor(L)$ the set of all left factors of words in L, i. e., $LeftFactor(L) = \{x \in A^* \mid \exists y \in L : x \leq y\}$. ■

Definition (4.2.5.2): Let A be a finite alphabet.

A language $L \subset A^*$ is called *strictly monogeneous* if $\exists$ words $u, v \in A^* : L \subset LeftFactor(uv^*)$.

A language $L \subset A^*$ is called *monogeneous* if it is equal to a finite union of strictly monogeneous languages, i. e., $L \subset \bigcup_{i=1,\ldots,k} LeftFactor(u_i v_i^*)$, where $\forall i \in \{1,\ldots,k\} : u_i, v_i \in A^*$. ■

Definition (4.2.5.3): Let (FPN, M_0) be a marked FIFO Petri Net. Let $p \in P$ be a place of FPN.

- p is called *structurally monogeneous*, if $\exists$ word $u_p \in A^*$ such that $\forall t \in T : F(t,p) \in u_p^*$.
- p is called *strictly monogeneous* if $L_I(FPN, M_0, p)$ is strictly monogeneous.
- p is called *monogeneous* if $L_I(FPN, M_0, p)$ is monogeneous.

(FPN, M_0) is called a *Monogeneous* (*Structurally Monogeneous*, *Strictly Monogeneous*, respectively) *FIFO Petri Net* if, and only if, each of its places is monogeneous (structurally monogeneous, strictly monogeneous, respectively). ■

Unfortunately, there exists no common understanding of these terms in the literature. In [Rou87] for example, the term *monogeneous* is used instead of *strictly monogeneous* and *semi-monogeneous* is used instead of *monogeneous*. Even more confusing, in [Sta83] and [Fin84] the term *monogeneous* is used for the weaker *structurally monogeneous*.

While undecidable in the general case, [Fin86] provides many sufficient and necessary conditions for a FIFO Petri Net to be monogeneous.

Linear FIFO Petri Nets

In this part, we follow the notation in [FR88].

Definition (4.2.5.4): Let A be a finite alphabet. A language $L \subset A^*$ is called *bounded* or *linear* if L is included in $a_1^* \ldots a_n^*$ for some $a_1, \ldots, a_n \in A$ with $\forall i \neq j : a_i \neq a_j$. ■

Definition (4.2.5.5): Let (FPN, M_0) be a marked FIFO Petri Net. Let $p \in P$ be a place of FPN. p is called *linear* if its input language is bounded.

(FPN, M_0) is called a *Linear FIFO Petri Net* (LFPN) if, and only if, each of its places is linear and has as its initial marking an element of a_1^*. ■

Definition (4.2.5.6): Let (FPN, M_0) be a marked LFPN with $FPN = (P, T, B, F, Q)$. Let SM be a set of markings over P. SM is called a *Structured Set of Terminal Markings* (SSTM) with respect to (FPN, M_0) if, and only if:

(i) membership in SM is decidable,

(ii) $M_0 \in SM$,

(iii) $\forall x, y \in T^* : (M_0(xy > M \wedge M_0(x > M' \wedge M \in SM) \Rightarrow M' \in SM$ (i. e., each marking reached on a path into SM must be in SM), and

(iv) $\forall x \in T^* : (M \in SM \wedge M(x^i > M_i, i \geq 1 \wedge M \leq M_1 \wedge M_1 \in SM) \Rightarrow \forall i \geq 1 : M_i \in SM$ (i. e., any sequence of transitions which when applied to a marking in SM terminates at another marking in SM and can be repeated indefinitely without leaving SM). ■

The notation $M \leq M_1$ relates to the definition of left factors. $M \leq M_1$ if, and only if, for all places $p \in P$ the marking M of p is a left factor of the marking M_1 of p.

Definition (4.2.5.7): Let (FPN, M_0) be a marked LFPN with $FPN = (P, T, B, F, Q)$. Let SM be a structured set of terminal markings over P. (FPN, M_0, SM) is called a *Linear FIFO Petri Net having a Structured Set of Terminal Markings* (SSTM–LFPN). The *set of firing sequences* (language) of (FPN, M_0, SM) is $FS(FPN, M_0, SM) = \{x \mid x \in T^*, M_0(x > M, M \in SM\}$. ■

The reachability tree for (FPN, M_0, SM) is simply the reachability tree for (FPN, M_0) pruned by truncating a path whenever it leaves SM. Therefore, the following holds for the reachability set of (FPN, M_0, SM):

$$RS(FPN, M_0, SM) = RS(FPN, M_0) \cap SM$$

Topologically Free Choice FIFO Petri Nets

In this part, we follow the notation in [FC88].

Definition (4.2.5.8): Let (FPN, M_0) be a marked FIFO Petri Net. Let $p \in P$ be a place of the FIFO Petri Net.

- The *input alphabet* of p is the set of all letters that appear in the valuation of at least one input arc of p.
- The *output alphabet* of p is the set of all letters appearing in the valuations of the output arcs.
- The *alphabet* of p, denoted by A_p, is the union of the input alphabet and the output alphabet.

■

Definition (4.2.5.9): Let (FPN, M_0) be a marked FIFO Petri Net. Let $p \in P$ be a place and $t \in T$ be a transition of the FIFO Petri Net. We define:

$$\begin{aligned} \Gamma(p) &= \{v \in T \mid B(p,v) \neq \lambda\} \\ \Gamma(t) &= \{v \in P \mid F(t,v) \neq \lambda\} \\ \Gamma^{-}(p) &= \{v \in T \mid F(v,p) \neq \lambda\} \\ \Gamma^{-}(t) &= \{v \in P \mid B(v,t) \neq \lambda\} \end{aligned}$$ ■

Definition (4.2.5.10): Let (FPN, M_0) be a marked FIFO Petri Net. (FPN, M_0) is called *normalized* if the following three conditions are satisfied:

(i) Each place $p \in P$ is balanced, i. e., the input alphabet is identical to the output alphabet.

(ii) $\forall p \in P \; \forall t \in \Gamma(p) \; : \; B(t,p) \in Q$, i. e., each place is semi-alphabetic.

(iii) $\forall p \in P \; : \; M_0(p) \in A_p^*$. ■

Definition (4.2.5.11): *Hack's condition* for *free choice Petri Nets* reads as follows: A place p in a Petri Net is free choice if. and only if. we have:

$$\mid \Gamma(p) \mid > 1 \Rightarrow \forall t \in \Gamma(p) \; : \; \Gamma^{-}(t) = \{p\}$$ ■

Definition (4.2.5.12): Let (FPN, M_0) be a marked FIFO Petri Net. (FPN, M_0) is called an *Extended Topologically Free Choice FIFO Petri Net* (ETFC-FPN) if. and only if, the following two conditions are satisfied:

- (FPN, M_0) is normalized.
- $\forall p \in P \; : \mid A_p \mid > 1 \Rightarrow p$ satisfies the Hack's condition. ■

4.3 Subclasses of Formalized Data Flow Diagrams

We assume that the reader is familiar with the concept of FDFD's given in [LWBL96] and [SB96a]. A short summary of [LWBL96] and definitions of Reduced Data Flow Diagrams (RDFD's) and persistent flow-free Reduced Data Flow Diagrams (PFF-RDFD's) is given in [SB96a] as well. Here, we will only provide three basic definitions related to FDFD's.

Definition (4.3.1): A *Formalized Data Flow Diagram* (FDFD) is a quintuple

$$FDFD = (B, FLOWNAMES, TYPES, P, F),$$

where B is a set of *bubbles*, $FLOWNAMES$ is a set of *flows*, $TYPES$ is a set of *types*, P is the set $\{persistent, consumable\}$ and $F = B \times FLOWNAMES \times TYPES \times B \times P$. The following notational convention for members from these domains is used: $b \in B, fn \in FLOWNAMES, T \in TYPES, p \in P, f \in F$. ■

Definition (4.3.2): A *firing sequence* (*computation sequence*) of an FDFD is a possibly infinite sequence $(b_i, a_i, j_i) \in B \times \{C, P\} \times \mathbb{N}, i \geq 0$, such that, if transition (b_i, a_i, j_i) is fired in state (bm, r, fs), then

$$(fs', r') = \begin{cases} (Consume(b_i))_{j_i}(fs, r), & \text{if } a_i = C \\ (Produce(b_i))_{j_i}(fs, r), & \text{if } a_i = P \end{cases}$$

$$bm'(b_i) = \begin{cases} working, & \text{if } a_i = C \\ idle, & \text{if } a_i = P \end{cases}$$

$$bm'(b) = bm(b) \quad \forall b \in B - \{b_i\}$$

and

$$(bm, r, fs) \rightarrow (bm', r', fs').$$

We introduce the notation $(bm, r, fs)[(b, a, j)]$ to indicate that transition (b, a, j) is fireable in state (bm, r, fs) and $(bm, r, fs)[(b, a, j)](bm', r', fs')$ to indicate that state (bm', r', fs') is reached upon the firing of transition (b, a, j) in state (bm, r, fs).

By induction, we extend this notation for firing sequences:

$$(bm_0, r_0, fs_0)[(b_1, a_1, j_1), \ldots, (b_{n-1}, a_{n-1}, j_{n-1}), (b_n, a_n, j_n)]$$

is used to indicate that transition (b_n, a_n, j_n) is fireable in state $(bm_{n-1}, r_{n-1}, fs_{n-1})$, given that

$$(bm_0, r_0, fs_0)[(b_1, a_1, j_1), \ldots, (b_{n-1}, a_{n-1}, j_{n-1})](bm_{n-1}, r_{n-1}, fs_{n-1})$$

holds. By analogy, we use

$$(bm_0, r_0, fs_0)[(b_1, a_1, j_1), \ldots, (b_n, a_n, j_n)](bm_n, r_n, fs_n)$$

to indicate that state (bm_n, r_n, fs_n) is reached upon the firing of the sequence $(b_1, a_1, j_1), \ldots, (b_n, a_n, j_n)$.

■

Definition (4.3.3): The *set of firing sequences* (*set of computation sequences, language*) of an FDFD, denoted by $FS(FDFD, \gamma_{initial})$, is the set containing all firing sequences that are possible for this FDFD, given $\gamma_{initial} = (bm_{initial}, r_{initial}, fs_{initial}) = (bm_0, r_0, fs_0)$, i. e.,

$$FS(FDFD, \gamma_{initial}) = \{s \mid s \in (B \times \{C, P\} \times I\!N)^* \wedge \gamma_{initial}[s]\}.$$

An element $s \in (B \times \{C, P\} \times I\!N)^*$ is said to be in the center of $(FDFD, \gamma_{initial})$, denoted by $C(FDFD, \gamma_{initial})$, if, and only if, $\gamma_{initial}[s]\gamma$ and $FS(FDFD, \gamma)$ is infinite. ■

We give the next definition in analogy to Definition (4.2.4.1):

Definition (4.3.4): For a given FDFD with initial state $\gamma_{initial} = (bm_{initial}, r_{initial}, fs_{initial})$, we define the following decidability problems:

Total Deadlock Problem (TDP): Is $FS(FDFD, \gamma_{initial})$ finite?

Partial Deadlock Problem (PDP): Is there a finite path in $(FDFD, \gamma_{initial})$ that can not be extended, i. e., does there exist an $s \in (B \times \{C, P\} \times I\!N)^*$ such that $\gamma_{initial}[s]\gamma$ where no transition $(b, a, j) \in (B \times \{C, P\} \times I\!N)$ is fireable in state γ?

Boundedness Problem (BP): Is $RS(FDFD, \gamma_{initial})$ finite?

Reachability Problem (RP): For a state γ, is $\gamma \in RS(FDFD, \gamma_{initial})$?

Quasi–Liveness Problem (QLP): $\forall (b, a, j) \in (B \times \{C, P\} \times I\!N)$, is there an $s \in (B \times \{C, P\} \times I\!N)^*$ such that $\gamma_{initial}[s, (b, a, j)]$?

Liveness Problem (LP): $\forall \gamma \in RS(FDFD, \gamma_{initial})$ $\forall (b, a, j) \in (B \times \{C, P\} \times I\!N)$, is there an $s \in (B \times \{C, P\} \times I\!N)^*$ such that $\gamma[s, (b, a, j)]$?

Center Problem (CP): Is there an algorithm that will generate a recursive representation of $C(FDFD, \gamma_{initial})$?

Regularity Problem (RegP): Is $FS(FDFD, \gamma_{initial})$ regular? ■

4.3.1 Monogeneous (PFF–)RDFD's

Our definitions of Monogeneous (PFF–)RDFD's are related to the definitions of monogeneous languages and Monogeneous FIFO Petri Nets as given in [Fin86] and [FR88], summarized in Subsection 4.2.5.

Definition (4.3.1.1): For each flow $f \in F$ of an FDFD, we define

$$I_f : (B \times \{C,P\} \times I\!N) \times (BubbleMode \times Read \times FlowState) \rightarrow OBJECTS \cup \{<>\}$$

such that

$$I_f((b,a,j),(bm,r,fs)) = \begin{cases} o, & \text{if } a = P \text{ and } (Produce(b))_j = Out(o,f,b)(fs,r) \\ <>, & \text{otherwise} \end{cases}$$

where $(b,a,j) \in (B \times \{C,P\} \times I\!N)$ is a transition and $(bm,r,fs) \in (BubbleMode \times Read \times FlowState)$ is a state of the FDFD.

By induction, we define I_f for firing sequences:

$$I_f : (B \times \{C,P\} \times I\!N)^{n+1} \times (BubbleMode \times Read \times FlowState) \rightarrow (OBJECTS \cup \{<>\})^*$$

such that

$$I_f(((b_0,a_0,j_0),\ldots,(b_{n-1},a_{n-1},j_{n-1}),(b_n,a_n,j_n)),(bm,r,fs)) =$$
$$I_f(((b_0,a_0,j_0),\ldots,(b_{n-1},a_{n-1},j_{n-1})),(bm,r,fs)) \circ I_f((b_n,a_n,j_n),(bm',r',fs')),$$

where $(bm,r,fs)[(b_0,a_0,j_0),\ldots,(b_{n-1},a_{n-1},j_{n-1})](bm',r',fs')$ and "$\circ$" means the concatenation of words.

Finally, the *input language* of a flow $f \in F$ of an FDFD with initial state $\gamma_{initial} = (bm_{initial}, r_{initial}, fs_{initial})$ is defined as

$$\begin{aligned} \tilde{I}_f &:= I_f(FS(FDFD,\gamma_{initial}),\gamma_{initial}) \\ &= \{I_f(s,\gamma_{initial}) \mid s \in FS(FDFD,\gamma_{initial})\} \end{aligned}$$ ■

Definition (4.3.1.2): Let $f \in F$ be a flow of an FDFD.

- f is called *structurally monogeneous* if $\exists\, u_f \in OBJECTS \cup \{<>\}$ $\forall (b,a,j) \in (B \times \{C,P\} \times I\!N)$ $\forall (bm,r,fs) \in (BubbleMode \times Read \times FlowState)$:
  $$I_f((b,a,j),(bm,r,fs)) = \begin{cases} u_f, & \text{if } a = P \text{ and } (Produce(b))_j = Out(u_f,f,b)(fs,r) \\ <>, & \text{otherwise} \end{cases}$$
- f is called *strictly monogeneous* if $\tilde{I}_f$ is strictly monogeneous.
- f is called *monogeneous* if $\tilde{I}_f$ is monogeneous. ■

A structurally monogeneous flow of an FDFD is more restricted than a structurally monogenous place of a FIFO Petri Net. In the FDFD, a single object $u_f \in OBJECTS$ (or nothing) is appended to the flow, while in the FIFO Petri Net an entire word $u_p \in A^*$ can be appended to the place. This

limited behavior of the FDFD is caused by the built-in restrictions on *Produce* (see [LWBL96]) that do not allow expressions such as $Out(u_f, f, X)(Out(u_f, f, X)(fs, r))$, i. e., each outflow f can be addressed at most once in a single *Produce* case of bubble X.

Definition (4.3.1.3): A (PFF-)RDFD is called a *Monogeneous* (*Structurally Monogeneous*, *Strictly Monogeneous*, respectively) (PFF-)RDFD if, and only if, each of its flows $f \in F$ is monogeneous (structurally monogeneous, strictly monogeneous, respectively). ■

Note that every Structurally Monogeneous (PFF-)RDFD is also a Strictly Monogeneous (PFF-) RDFD, which is also a Monogeneous (PFF-)RDFD, i. e., Monogeneous (PFF-)RDFD's are the most general of these subclasses. If we state that a condition holds for Monogeneous (PFF-)RDFD's this obviously includes Structurally Monogeneous (PFF-)RDFD's and Strictly Monogeneous (PFF-)RDFD's.

Example (4.3.1.4): This example of an PFF-RDFD presents a simple communication protocol. Each participant, A and B, can initiate the communication but then has to wait for an acknowledgement from the other participant that matches its own message. It should be obvious that in this example we always have $Head(fs(BA)) = Head(fs(last_A))$ if both are not $\perp$, and $Head(fs(AB)) = Head(fs(last_B))$ if both are not $\perp$. In a system where erraneous channels are modeled instead of flows AB and BA, the current specification of bubbles A and B will most likely produce several deadlock states.
The mappings *Enabled*, *Consume*, and *Produce* for the FDFD shown in Figure 4.1 are defined as:

$$
\begin{aligned}
&Enabled(A) = \lambda fs\ .\\
&\quad (\neg IsEmpty(init_A) \wedge Head(fs(init_A)) = a)\\
&\quad \vee(\neg IsEmpty(BA) \wedge Head(fs(BA)) = a\\
&\qquad \wedge \neg IsEmpty(last_A) \wedge Head(fs(last_A)) = a)\\
&\quad \vee(\neg IsEmpty(BA) \wedge Head(fs(BA)) = b\\
&\qquad \wedge \neg IsEmpty(last_A) \wedge Head(fs(last_A)) = b)\\
&Enabled(B) = \lambda fs\ .\\
&\quad (\neg IsEmpty(init_B) \wedge Head(fs(init_B)) = a)\\
&\quad \vee(\neg IsEmpty(AB) \wedge Head(fs(AB)) = a\\
&\qquad \wedge \neg IsEmpty(last_B) \wedge Head(fs(last_B)) = a)\\
&\quad \vee(\neg IsEmpty(AB) \wedge Head(fs(AB)) = b\\
&\qquad \wedge \neg IsEmpty(last_B) \wedge Head(fs(last_B)) = b)
\end{aligned}
$$

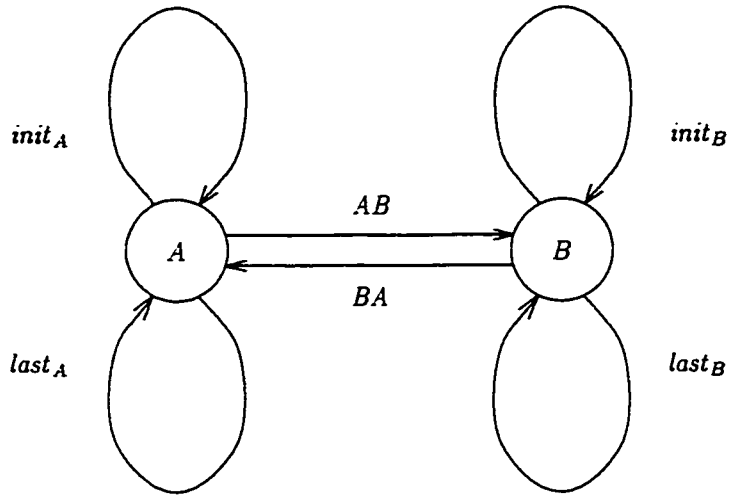

Figure 4.1: Example of a Strictly Monogeneous PFF-RDFD.

Transition

$Consume(A) = \lambda(fs, r)$.

{**if** $(\neg IsEmpty(init_A) \wedge Head(fs(init_A)) = a)$

then $In(init_A, A)(fs, r)$ (A, C, 1)

fi,

if $(\neg IsEmpty(BA) \wedge Head(fs(BA)) = a$

$\wedge \neg IsEmpty(last_A) \wedge Head(fs(last_A)) = a)$

then $In(BA, A)(In(last_A, A)(fs, r))$ (A, C, 2)

fi.

if $(\neg IsEmpty(BA) \wedge Head(fs(BA)) = b$

$\wedge \neg IsEmpty(last_A) \wedge Head(fs(last_A)) = b)$

then $In(BA, A)(In(last_A, A)(fs, r))$ (A, C, 3)

fi

}

$Consume(B) = \lambda(fs, r)$.

{**if** $(\neg IsEmpty(init_B) \wedge Head(fs(init_B)) = a)$

then $In(init_B, B)(fs, r)$ (B, C, 1)

fi.

if $(\neg IsEmpty(AB) \wedge Head(fs(AB)) = a$

$\wedge \neg IsEmpty(last_B) \wedge Head(fs(last_B)) = a)$

then $In(AB, B)(In(last_B, B)(fs, r))$ (B, C, 2)

fi,

if $(\neg IsEmpty(AB) \wedge Head(fs(AB)) = b$

$\wedge \neg IsEmpty(last_B) \wedge Head(fs(last_B)) = b)$

then $In(AB, B)(In(last_B, B)(fs, r))$ (B, C, 3)

fi

}

$Produce(A) = \lambda(fs, r)$.

{**if** $r(A)(init_A) = a$

then $Out(a, AB, A)(Out(a, last_A, A)(fs, r))$ (A, P, 1)

fi,

if $r(A)(BA) = a \wedge r(A)(last_A) = a$

then $Out(b, AB, A)(Out(b, last_A, A)(fs, r))$ (A, P, 2)

fi,

if $r(A)(BA) = b \wedge r(A)(last_A) = b$

then $Out(a, AB, A)(Out(a, last_A, A)(fs, r))$ (A, P, 3)

fi

}

$Produce(B) = \lambda(fs, r)$.

{**if** $r(B)(init_B) = a$

then $Out(a, BA, B)(Out(a, last_B, B)(fs, r))$ (B, P, 1)

fi,

if $r(B)(AB) = a \wedge r(B)(last_B) = a$

then $Out(b, BA, B)(Out(b, last_B, A)(fs, r))$ (B, P, 2)

fi,

if $r(B)(BA) = b \wedge r(A)(last_B) = b$

then $Out(a, BA, B)(Out(a, last_B, B)(fs, r))$ (B, P, 3)

fi

}

Initially, $init_A$ and $init_B$ contain an a. All other flows are empty. Valid firing sequences are, for example, $(A, C, 1), (A, P, 1), (B, C, 1), (B, P, 1)$.

$(A,C,2),(A,P,2),(B,C,2),(B,P,2),(A,C,3),(A,P,3),(B,C,3),(B,P,3),$

$(A,C,2),(A,P,2),(B,C,2),(B,P,2),(B,C,3),(B,P,3),(A,C,3),(A,P,3),\ldots$

and

$(B,C,1),(A,C,1),(A,P,1),(B,P,1),$

$(B,C,2),(B,P,2),(A,C,2),(A,P,2),(A,C,3),(B,C,3),(B,P,3),(A,P,3),$

$(A,C,2),(B,C,2),(B,P,2),(A,P,2),(B,C,3),(A,C,3),(B,P,3),(A,P,3),\ldots$.

We have $\tilde{I}_{init_A} = \tilde{I}_{init_B} = \{a\}$ and $\tilde{I}_{AB} = \tilde{I}_{BA} = \tilde{I}_{last_A} = \tilde{I}_{last_B} = LeftFactor((ab)^*)$. Thus, flows $init_A$ and $init_B$ are structurally monogeneous, and flows AB, BA, $last_A$, and $last_B$ are strictly monogeneous. Overall, the PFF–RDFD is strictly monogeneous. ■

Theorem (4.3.1.5): Every Monogeneous (Structurally Monogeneous, Strictly Monogeneous, respectively) PFF–RDFD can be simulated by a Monogeneous (Structurally Monogeneous, Strictly Monogeneous, respectively) FIFO Petri Net with respect to an isomorphism h.

Proof: In [SB96a] it has been shown that every PFF–RDFD can be simulated by a FIFO Petri Net with respect to an isomorphism h. Therefore, we only have to show that this isomorphism h maps every monogeneous (structurally monogeneous, strictly monogeneous, respectively) flow $f \in F$ of the PFF–RDFD to a monogeneous (structurally monogeneous, strictly monogeneous, respectively) place of the FIFO Petri Net.

First, we want to recall from [SB96a] that the set of places P_{FPN} of the related FIFO Petri Net can be split into three disjoint subsets, (i) representing the flows of the PFF–RDFD, (ii) the *idle* working mode of the bubble, and (iii) the *working* working mode (including the values that have been read) of the bubble, i. e.,

$$\begin{aligned} P_{FPN} = \{f_1,\ldots,f_f\} & \\ & \cup\{b_{1,idle},\ldots,b_{b,idle}\} \\ & \cup \bigcup_{i\in\{1,\ldots,b\}} \Big(\{b_{i,working:1} \mid Consume(b_i) \text{ is of type } C_1\} \\ & \qquad \cup\{b_{i,working:1},\ldots,b_{i,working:m_i} \mid Consume(b_i) \text{ is of type } C_2, \\ & \qquad\quad m_i = (\#\text{ of cases in } Consume(b_i))\} \Big) \end{aligned}$$

Now, we consider each of the subsets of places in P_{FPN}, with h given as in Theorem (3.1.1) in [SB96a]:

(i) $p \in \{f_1,\ldots,f_f\}$:

Since $M_{FPN}(p) = fs(p)$ by definition, the contents of each place of the FIFO Petri Net is identical to the contents of the corresponding flow of the PFF–RDFD. Also, a new value is appended to

place p if, and only if, the related value is appended to the corresponding flow. Hence, since flow p is monogeneous (structurally monogeneous, strictly monogeneous, respectively), place p is monogeneous (structurally monogeneous, strictly monogeneous, respectively), too.

(ii) $p \in \{b_{1,idle}, \ldots, b_{b,idle}\}$:

The only value that is appended to place p is I. Therefore, p is structurally monogeneous (which implies that it is strictly monogeneous and monogeneous).

(iii) $p \in \{b_{1,working:1}, \ldots, b_{1,working:m_1}, \ldots, b_{b,working:1}, \ldots, b_{b,working:m_b}\}$:

The only value that is appended to place p is W. Therefore, p is structurally monogeneous (which implies that it is strictly monogeneous and monogeneous).

So, since the PFF-RDFD is monogeneous (structurally monogeneous, strictly monogeneous, respectively), i. e., each of its flows is monogeneous (structurally monogeneous, strictly monogeneous, respectively), the FIFO Petri Net is monogeneous (structurally monogeneous, strictly monogeneous, respectively), too. ■

Example (4.3.1.6): The previous Theorem does not hold in general for RDFD's with persistent flows. Consider the RDFD given in Figure 4.2.

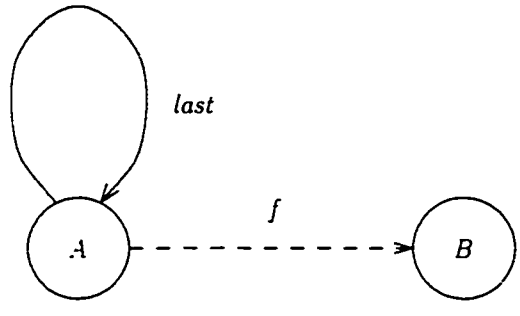

Figure 4.2: Example of a Strictly Monogeneous RDFD with Persistent Flow.

The mappings *Enabled*, *Consume*, and *Produce* are specified as follows:

$$Enabled(A) = \lambda fs\ .$$
$$(\neg IsEmpty(last) \wedge Head(fs(last)) = 0)$$
$$\vee(\neg IsEmpty(last) \wedge Head(fs(last)) = 1)$$
$$Enabled(B) = \lambda fs\ .$$
$$Head(fs(f)) = 0 \vee Head(fs(f)) = 1$$

$Consume(A) = \lambda(fs, r)$.

{**if** $(\neg IsEmpty(last) \wedge Head(fs(last)) = 0)$

then $In(last, A)(fs, r)$

fi,

if $(\neg IsEmpty(last) \wedge Head(fs(last)) = 1)$

then $In(last, A)(fs, r)$

fi

}

$Consume(B) = \lambda(fs, r)$.

{**if** $Head(fs(f)) = 0$

then $In(f, B)(fs, r)$

fi,

if $Head(fs(f)) = 1$

then $In(f, B)(fs, r)$

fi

}

$Produce(A) = \lambda(fs, r)$.

{**if** $r(A)(last) = 0$

then $Out(1, f, A)(Out(1, last, A)(fs, r))$

fi,

if $r(A)(last) = 1$

then $Out(0, f, A)(Out(0, last, A)(fs, r))$

fi

}

$Produce(B) = \lambda(fs, r) . \{(fs, [b_i \mapsto \lambda f . \bot]r)\}$

Initially, f and *last* contain a 0. Obviously, $\tilde{I}_{last} = \tilde{I}_f = LeftFactor((01)^*)$, i. e., flows *last* and f are strictly monogeneous. Overall, the RDFD is strictly monogeneous. The equivalent FIFO Petri Net constructed according to [SB96a] is given in Figure 4.3. Since $L_I(FPN, M_0, last) = LeftFactor((01)^*)$, place *last* is strictly monogeneous, but since $L_I(FPN, M_0, f) = LeftFactor((0^+1^+)^*)$, place f is not strictly monogeneous (it is not even monogeneous). ■

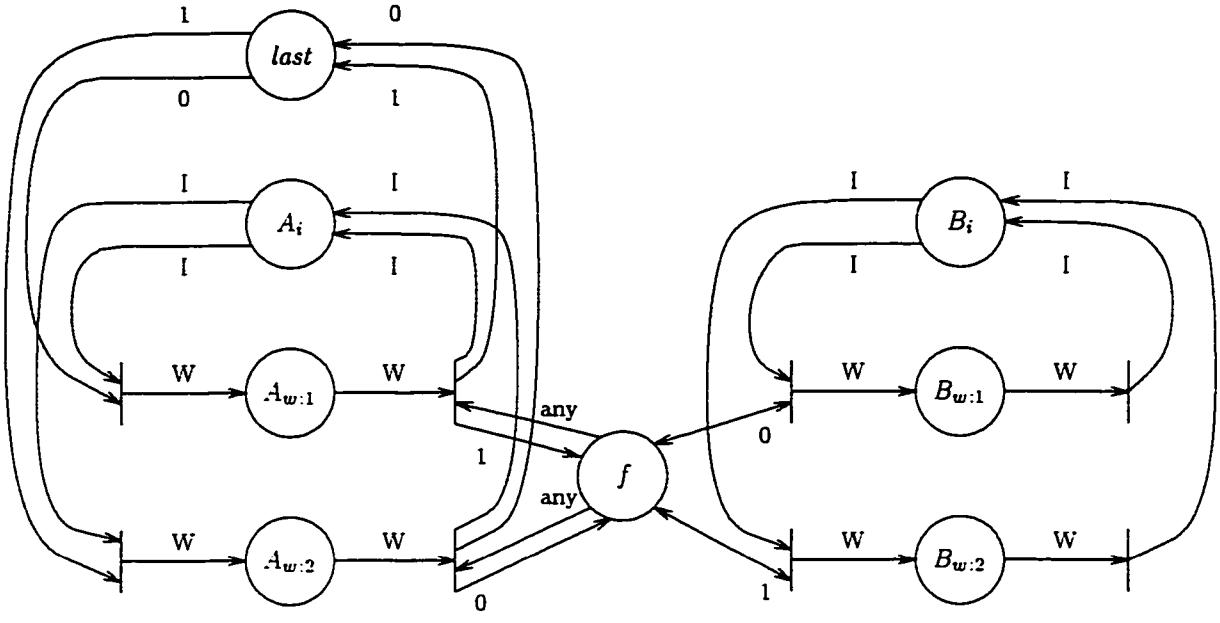

Figure 4.3: FIFO Petri Net Equivalent to a Monogeneous RDFD with Persistent Flow.

Corollary (4.3.1.7): The following problems are decidable for Monogeneous (Structurally Monogeneous, Strictly Monogeneous, respectively) PFF-RDFD's: TDP, PDP, BP, RP, QLP, LP, and RegP. The center of a Monogeneous (Structurally Monogeneous, Strictly Monogeneous, respectively) PFF-RDFD is effectively realizable, i. e., the CP is decidable.

Proof: All problems are decidable with respect to Monogeneous (Structurally Monogeneous, Strictly Monogeneous, respectively) FIFO Petri Nets ([Fin86], [FR88]). We have shown that there exists an isomorphism h between Monogeneous (Structurally Monogeneous, Strictly Monogeneous, respectively) PFF-RDFD's and Monogeneous (Structurally Monogeneous, Strictly Monogeneous, respectively) FIFO Petri Nets.

- According to the note following Theorem (4.2.2.5), h preserves TDP, PDP, BP, RP, and LP ([KM82]).
- QLP is decidable since LP is decidable with $\gamma = \gamma_{initial}$.
- CP is decidable for Monogeneous (Structurally Monogeneous, Strictly Monogeneous, respectively) FIFO Petri Nets ([FR88]).

  Since ρ is bijective, $x \in C(PFF\text{-}RDFD, \gamma_{initial}) \Leftrightarrow \rho(x) \in C(FPN, M_0)$ where $M_0 = \rho(\gamma_{initial})$.

- RegP is decidable for Monogeneous (Structurally Monogeneous, Strictly Monogeneous, respectively) FIFO Petri Nets ([FR88]).

  Since τ is bijective, $FS(PFF\text{-}RDFD, \gamma_{initial})$ is regular $\Leftrightarrow$ $\tau(FS(FPN, M_0))$ is regular where $M_0 = \rho(\gamma_{initial})$. ■

Without giving a definition of a Petri Net (see [Pet81], for example), we state the next corollary:

Corollary (4.3.1.8): Every Monogeneous (Structurally Monogeneous, Strictly Monogeneous, respectively) PFF–RDFD can be simulated by a deterministic Petri Net.

Proof: In [Sta83] and [Fin84] it is shown that every Structurally Monogeneous FIFO Petri Net can be simulated by a labelled Petri Net.[4] In [FR88] it is shown that every Monogeneous FIFO Petri Net can be simulated by a deterministic Petri Net. Therefore, we can simulate any given Monongeneous PFF–RDFD by a Monogeneous FIFO Petri Net which is then simulated by a Petri Net. ■

The key point in this series of simulations is that every solvable decidability problem for Petri Nets remains decidable for Monogeneous FIFO Petri Nets ([Fin84]) and for Monogeneous PFF–RDFD's. Solution techniques such as the reachability tree and matrix equation approaches can be used to determine other properties such as safeness, boundedness, conservation, and coverability for Petri Nets ([Pet81]). Therefore, we immediately have solution techniques to answer related questions for Monogeneous PFF–RDFD's.

4.3.2 Linear RDFD's

Our definitions of Linear RDFD's are related to the definitions of Linear FIFO Petri Nets as given in [FR88], summarized in Subsection 4.2.5.

Definition (4.3.2.1): Let $f \in F$ be a flow of an FDFD. f is called *linear* if its input language is bounded. ■

Definition (4.3.2.2): An RDFD is called a *Linear RDFD* (L–RDFD) if, and only if, each of its flows is linear and has as its initial flow state an element of a_1^*, where $a_1 \in OBJECTS$. ■

[4] Note that in these two references the term *monogeneous* is used instead of the term *structurally monogeneous* which is used within this paper.

Definition (4.3.2.3): Let $\gamma_0 \in \Gamma$ be the initial state of an L-RDFD. Let $S\Gamma$ be a set of states over $\Gamma = (BubbleMode \times Read \times FlowState)$. $S\Gamma$ is called a *Structured Set of Terminal States* (SSTS) with respect to $(L\text{-}RDFD, \gamma_0)$ if, and only if:

(i) membership in $S\Gamma$ is decidable,

(ii) $\gamma_0 \in S\Gamma$,

(iii) $\forall x, y \in (B \times \{C, P\} \times I\!N)^* : (\gamma_0[x, y]\gamma \wedge \gamma_0[x]\gamma' \wedge \gamma \in S\Gamma) \Rightarrow \gamma' \in S\Gamma$ (i. e., each state reached on a path into $S\Gamma$ must be in $S\Gamma$), and

(iv) $\forall x \in (B \times \{C, P\} \times I\!N)^* : (\gamma \in S\Gamma \wedge \gamma[x^i]\gamma_i, i \geq 1 \wedge \gamma \leq \gamma_1 \wedge \gamma_1 \in S\Gamma) \Rightarrow \forall i \geq 1 : \gamma_i \in S\Gamma$ (i. e., any sequence of transitions which when applied to a state in $S\Gamma$ terminates at another state in $S\Gamma$ and can be repeated indefinitely without leaving $S\Gamma$). ■

Definition (4.3.2.4): Let $\gamma_1 = (bm_1, r_1, fs_1), \gamma_2 = (bm_2, r_2, fs_2) \in \Gamma$ be states of an FDFD. We say that $\gamma_1 \leq \gamma_2$ if, and only if, the following three conditions hold:

- $\forall b \in B : bm_1(b) = bm_2(b)$
- $\forall b \in B \forall f \in F : r_1(b)(f) = r_2(b)(f)$
- $\forall f \in F : fs_1(f) \leq fs_2(f)$, i. e., $fs_1(f)$ is a left factor of $fs_2(f)$. ■

Definition (4.3.2.5): Let $\gamma_0 \in \Gamma$ be the initial state of an L-RDFD. Let $S\Gamma$ be a set of states over $\Gamma = (BubbleMode \times Read \times FlowState)$. $(L\text{-}RDFD, \gamma_0, S\Gamma)$ is called a *Linear RDFD having a Structured Set of Terminal States* (SSTS-L-RDFD). The *set of firing sequences* of $(L\text{-}RDFD, \gamma_0, S\Gamma)$ is $FS(L\text{-}RDFD, \gamma_0, S\Gamma) = \{s \mid s \in (B \times \{C, P\} \times I\!N)^* \wedge \gamma_0[s]\gamma \wedge \gamma \in S\Gamma\}$. ■

Theorem (4.3.2.6): Every (SSTS-)L-RDFD (with a Structured Set of Terminal States $S\Gamma$) with initial state $\gamma_{initial}$ can be simulated by a Linear FIFO Petri Net (with a Structured Set of Terminal Markings $\rho(S\Gamma)$) with respect to an isomorphism h.

Proof: Similar to the proof of Theorem (4.3.1.5), we distinguish among four different types of places in P_{FPN}:

(i) $p \in \{f_1, \ldots, f_f\} \wedge Consumable(p)$:

Since $M_{FPN}(p) = fs(p)$ by definition, the contents of each place of the FIFO Petri Net is identical

to the contents of the corresponding flow of the L–RDFD. Also, a new value is appended to place p if, and only if, the related value is appended to the corresponding flow. Hence, since flow p is linear, place p is linear, too.

(ii) $p \in \{f_1, \ldots, f_f\} \wedge \neg Consumable(p)$:

Since $M_{FPN}(p) = fs(p)$ by definition, the contents of each place of the FIFO Petri Net is identical to the contents of the corresponding flow of the L–RDFD. A new value is appended to place p if, and only if, one of two possible cases occurs:

(a) The related value is appended to the corresponding flow. Then, since flow p is linear, place p is linear, too.

(b) A value is read from the corresponding persistent flow (but it is not removed from this flow). This relates to removing the head element and appending the new value (which is the same as the value which has been removed) to this place upon firing of a transition of the FIFO Petri Net. Since our mapping from RDFD's to FIFO Petri Nets guarantees that places representing persistent flows contain exactly one token at a time, this new value appended to place p is automatically the head element of this place. Therefore, if in the L–RDFD the word $\ldots a_i^{n_i} \ldots$ occurs as input to flow p, the word $\ldots a_i^{n_{i1}} a_i a_i^{n_{i2}} \ldots$, where $n_{i1} \geq 1, n_i = n_{i1} + n_{i2}$, will occur as input to place p in the FIFO Petri Net. Hence, place p is linear.

(iii) $p \in \{b_{1,idle}, \ldots, b_{b,idle}\}$:

The only value that is appended to place p is I. The input language of p is I^* with initial marking I. Therefore, p is linear.

(iv) $p \in \{b_{1,working:1}, \ldots, b_{1,working:m_1}, \ldots, b_{b,working:1}, \ldots, b_{b,working:m_b}\}$:

The only value that is appended to place p is W. The input language of p is W^* with initial marking $<>$. Therefore, p is linear.

So, since the L–RDFD is linear, i. e., each of its flows is linear, the FIFO Petri Net is linear, too. Now, we still have to show that $\rho(S\Gamma)$ is a Structured Set of Terminal Markings with respect to $(FPN, \rho(\gamma_{initial}))$. Since ρ is bijective, $\gamma \in S\Gamma \Leftrightarrow \rho(\gamma) \in \rho(S\Gamma)$. Since τ is bijective, $s \in FS(L\text{-}RDFD, \gamma_{initial}, S\Gamma) \Leftrightarrow \tau(s) \in FS(FPN, \rho(\gamma_{initial}), \rho(S\Gamma))$. Therefore, $\rho(S\Gamma)$ is a SSTM of the FIFO Petri Net since $S\Gamma$ is a SSTS of the L–RDFD. ■

Formally, we can incorporate the notation of a Structured Set of Terminal States $S\Gamma$ into the transitions rules (see [LWBL96], [SB96a]) that are allowed between configurations of FDFD's. The modified transition rules now read as follows:

$$\frac{\begin{array}{c} bm(b) = idle, \\ Enabled(b)(fs) = true, \\ bm' = [b \mapsto working]bm, \\ (fs', r') \in Consume(b)(fs, r) \\ (bm', r', fs') \in S\Gamma \end{array}}{(bm, r, fs) \longrightarrow (bm', r', fs')}$$

and

$$\frac{\begin{array}{c} bm(b) = working, \\ bm' = [b \mapsto idle]bm, \\ (fs', r') \in Produce(b)(fs, r) \\ (bm', r', fs') \in S\Gamma \end{array}}{(bm, r, fs) \longrightarrow (bm', r', fs')}$$

Of course, it must be decidable whether $(bm', r', fs') \in S\Gamma$ holds.

There are two obvious advantages of having a SSTS for FDFD's (and not only for L-RDFD's):

- A computerized evaluation of a given FDFD, for example by using the software described in [Wah95], may be restricted to those states that are of particular interest to the system analyst.
- The introduction of an SSTS is an additional approach to modify the qualitative behavior of an FDFD. For example, consider an FDFD where a communication protocol with erraneous channels has been modeled. Assume we also have been able to identify a set of error states, ES. Then, if we want to analyze a similar communication protocol where no erraneous channels occur, we do not have to modify the FDFD itself, but just have to introduce the SSTS $S\Gamma = RS(FDFD, \gamma_{initial}) - ES$, such that it is impossible for the system to enter any of the error states.

Corollary (4.3.2.7): The following problems are decidable for (SSTS-)L-RDFD's: TDP, PDP, BP, RP, and QLP.

Proof: All problems are decidable with respect to (SSTM-)LFPN's ([FR88]). We have shown that there exists an isomorphism h between (SSTS-)L-RDFD's and SSTM-LFPN's.

- According to the note following Theorem (4.2.2.5), h preserves TDP, PDP, BP, and RP ([KM82]).
- QLP is decidable for (SSTM–)LFPN's ([FR88]).
  Since τ and ρ are bijective, $\forall(b,a,j) \in (B_{RDFD} \times \{C,P\} \times I\!N)\ \ \exists s \in (B_{RDFD} \times \{C,P\} \times I\!N)^* :$ $\gamma_{initial}[s,(b,a,j)]$ holds $\Leftrightarrow \forall t \in T_{FPN}\ \ \exists \tilde{x} \in T_{FPN}^* : M_0(\tilde{x}t >$ holds, where $M_0 = \rho(\gamma_{initial})$, $t = \tau((b,a,j))$, and $\tilde{x} = \tau(s)$. ■

4.3.3 Topologically Free Choice RDFD's

Our definitions of Topologically Free Choice RDFD's are related to the definitions of Topologically Free Choice FIFO Petri Nets as given in [FC88], summarized in Subsection 4.2.5.

Definition (4.3.3.1): Let $f \in F$ be a flow of an FDFD.
The *output alphabet* AO_f of a flow f is defined as

$$AO_f = \{o \mid \exists b \in B\ \exists j \in I\!N : (Consume(b))_j = \ldots Head(fs(f)) = o \ldots\}.$$

The *input alphabet* AI_f of a flow f is defined as

$$AI_f = \{i \mid \exists b \in B\ \exists j \in I\!N : (Produce(b))_j = \ldots Out(i,f,b) \ldots\}.$$

The *alphabet* A_f of a flow f is defined as $A_f = AO_f \cup AI_f$. ■

The output alphabet AO_f is a subset of *OBJECTS* that might be read from a flow f in accordance with the mapping *Consume*. The input alphabet AI_f is a subset of *OBJECTS* that might be written to a flow f in accordance with the mapping *Produce*. These definitions are only related to the static structure of the FDFD. It is not necessarily required that all *OBJECTS* $a \in A_f$ will actually appear on this flow for any firing sequence or any initial state $\gamma_{initial}$.

Definition (4.3.3.2): The set of flows F of an FDFD with initial state $\gamma_{initial} = (bm_{initial}, r_{initial}, fs_{initial})$ is called *normalized*, if the following two conditions are satisfied:

- each flow $f \in F$ is balanced, i. e., $AI_f = AO_f = A_f$ (the input alphabet is equal to the output alphabet).
- $\forall f \in F : fs_{initial}(f) \in A_f^*$. ■

The definition of a normalized FIFO Petri Net requires that each place $p \in P$ is semi–alphabetic, i. e., at most one element of the alphabet A is consumed from p in each step. However, this is already part of our definition of RDFD's which states that for a bubble $b \in B$, the mappings $Enabled(b)$ and $Consume(b)$ only make use of the head element of a flow $f \in Inputs(b)$. Hence, each flow is semi–alphabetic in an RDFD. Actually, it is even alphabetic since a similiar restriction prevents $Produce(b)$ to write more than one element at a time to a flow $f \in Outputs(b)$.

In particular, the restriction to a set of normalized flows is no restriction of the power of RDFD's but guarantees, a priori, that there will never be an object $o \in OBJECTS$ which can not even potentially be removed from a flow $f \in F$, in at least one $Consume(b)$ case. Of course, it must hold that all other flows in this $Consume(b)$ case have the appropriate head element before this object actually can be removed.

In analogy to Hack's definition for free choice Petri Nets we extend this definition for RDFD's:

Definition (4.3.3.3): Let $f \in F$ be a flow of an FDFD and $b \in B$ the bubble where $f \in Inputs(b)$. f is called *free choice* (it satisfies the Hack condition) if, and only if, it fulfills one of two possible conditions:

- A statement of the form "$\ldots Head(fs(f)) \ldots$" occurs only in one single case in *Enabled/Consume* of bubble b, or
- for all cases in *Enabled/Consume* of bubble b that contain "$\ldots Head(fs(f)) \ldots$", f is the only flow that is used for this case (throughout the statement we have "$\neg IsEmpty(fs(f)) \wedge Head(fs(f)) = i$" for some i's). ■

The main idea of this definition is to allow only controlled conflict. In general, conflict occurs when several cases in bubble $b \in B$ could potentially read from the same flow $f \in F$. By the definition of free choice RDFD's, if a flow f occurs in several cases in *Enabled/Consume* of bubble b (potential conflict), then it is the only flow accessed in any of these cases. Therefore, all of the conflicting cases that require "$Head(fs(f)) = i$" are simultaneously activated, or none of them is activated since the flow is empty. This allows the choice (conflict resolution) to be made freely which case is to be selected. It does not depend on the presence of other *OBJECTS* on other flows.

Definition (4.3.3.4): An RDFD is called an *Extended Topologically Free Choice RDFD* (ETFC–RDFD), if, and only if, the following two conditions are satisfied:

- the set of flows F is normalized, and
- $\forall f \in F : \mid A_f \mid > 1 \Rightarrow f$ is free choice (it satisfies the Hack condition). ■

Since the set of flows is normalized, there exists for each flow $f \in F$ of an FDFD and $b \in B$ the bubble where $f \in \mathit{Inputs}(b)$ at least one case in *Enabled/Consume* that can make use of the head element of f, thus potentially go from *idle* to *working*, provided f and, if f occurs only in a single case, all other flows that occur in this case, are not empty.

Unfortunately, our construction of FIFO Petri Nets based on a given EFCT–RDFD fails to provide an EFCT–FIFO Petri Net. The problem is structurally inherited from the definition of the isomorphism h. For each bubble in the RDFD, we introduce additional places in the FIFO Petri Net to store the bubble's working mode and the values that have been read ([SB96a]). The place of the FIFO Petri Net that represents the *idle* working mode of the RDFD causes the problem since it typically is not the only input to serveral transitions of the FIFO Petri Net. Consider the following example:

Example (4.3.3.5): A simple FDFD with only two bubbles A and B connected by a flow f.

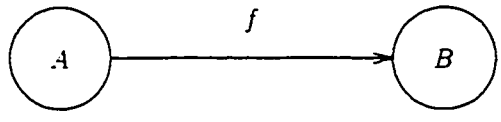

Figure 4.4: EFCT–RDFD.

The mappings *Enabled*, *Consume*, and *Produce* for the FDFD shown in Figure 4.4 are specified as follows:

$$\mathit{Enabled}(A) = \lambda fs \,.\, \mathit{true}$$

$$\mathit{Enabled}(B) = \lambda fs \,.$$
$$(\neg \mathit{IsEmpty}(f) \wedge \mathit{Head}(fs(f)) = 0)$$
$$\vee(\neg \mathit{IsEmpty}(f) \wedge \mathit{Head}(fs(f)) = 1)$$

$$\mathit{Consume}(A) = \lambda(fs, r) \,.\, \{(fs, r)\}$$

$$\mathit{Consume}(B) = \lambda(fs, r) \,.$$
$$\{\textbf{if } (\neg \mathit{IsEmpty}(f) \wedge \mathit{Head}(fs(f)) = 0)$$
$$\textbf{then } \mathit{In}(f, B)(fs, r)$$

fi,

if $(\neg IsEmpty(f) \wedge Head(fs(f)) = 1)$

then $In(f, B)(fs, r)$

fi

}

$Produce(A) = \lambda(fs, r)$.

$\{Out(0, f, A)(fs.r),$

$Out(1, f, A)(fs, r)\}$

$Produce(B) = \lambda(fs, r)$. $\{(fs, [B \mapsto \lambda f$. $\perp]r)\}$

Initially, flow f is empty. According to [SB96a]. the given RDFD transforms into the following marked FIFO Petri Net $FPN = ((P_{FPN}, T_{FPN}, B_{FPN}, F_{FPN}, Q_{FPN}), M_{0,FPN})$:

$P_{FPN} = \{f\} \cup \{A_i, A_{w:1}, B_i, B_{w:1}, B_{w:2}\}$

$T_{FPN} = \{C_{A1}, C_{B1}, C_{B2}\} \cup \{P_{A1}, P_{A2}, P_{B1}, P_{B2}\}$

The initial marking $M_{0,FPN}$ is such that:

$M_{0,FPN}(A_i) = M_{0,FPN}(B_i) = I$

$M_{0,FPN}(A_{w:1}) = M_{0,FPN}(B_{w:1}) = M_{0,FPN}(B_{w:2}) = <>$

$M_{0,FPN}(f) = <>$

B_{FPN}, F_{FPN}, and Q_{FPN} can be gained from Figure 4.5.

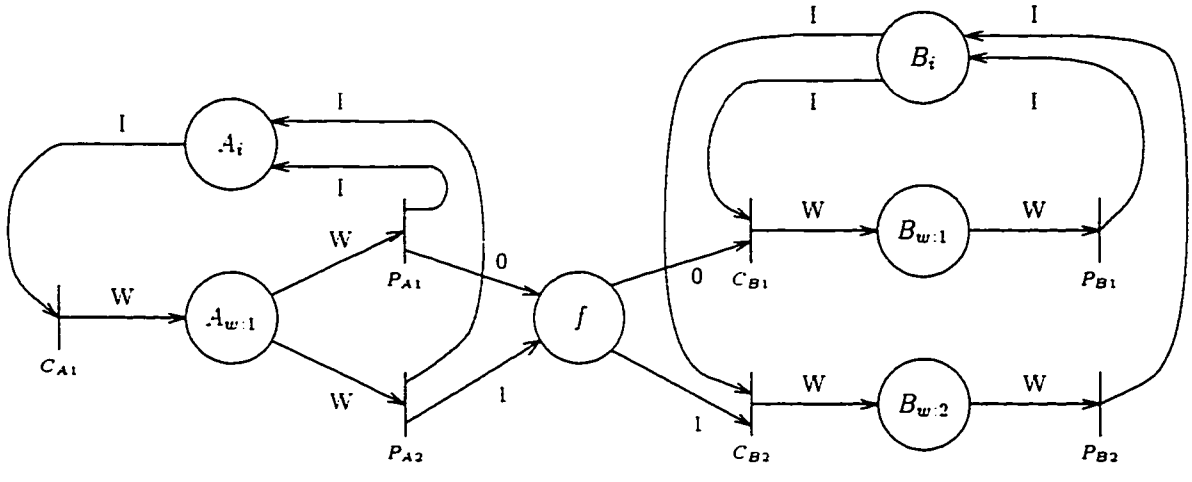

Figure 4.5: FIFO Petri Net.

The resulting FIFO Petri Net is normalized. We have:

$A_f = \{0, 1\}$

$A_{A_i} = A_{B_i} = \{I\}$

$A_{A_{w:1}} = A_{B_{w:1}} = A_{B_{w:2}} = \{W\}$

Each place is semi-alphabetic and the condition for the initial marking $M_{0,FPN}$ is fulfilled. Since $\mid A_f \mid = 2$, we have to verify that f satisfies the Hack condition. Unfortunately, it does not. We have $\Gamma(f) = \{C_{B_1}, C_{B_2}\}$ and $\mid \Gamma(f) \mid = 2 > 1$, but $\Gamma^{-1}(C_{B_1}) = \Gamma^{-1}(C_{B_2}) = \{f, B_i\} \neq \{f\}$. ■

4.4 Summary

The basic idea of this article was not to define completely new subclasses of RDFD's, but to extend known subclasses of FIFO Petri Nets towards RDFD's. Once defined, we have seen that Monogeneous PFF-RDFD's and Linear RDFD's are related to Monogeneous FIFO Petri Nets and Linear FIFO Petri Nets, respectively, through isomorphisms. These isomorphisms maintain solutions of decidability problems, thus allowing us to answer problems such as TDP, PDP, BP, RP, QLP, LP, RegP, and CP for Monogeneous PFF-RDFD's and problems such as TDP, PDP, BP, RP, and QLP for Linear RDFD's, based on methods and algorithms already available for FIFO Petri Nets. Unfortunately, our mapping from RDFD's to FIFO Petri Nets fails for ETFC-RDFD's. We are working on a different homomorphism h' between ETFC-RDFD's and ETFC-FIFO Petri Nets that hopefully will allow us to answer decidability problems for ETFC-RDFD's based on their solution for ETFC-FIFO Petri Nets.

Future work is expected to move in the following directions: It is desirable to identify further subclasses of RDFD's that allow the solution of (some) decidability questions. These new subclasses of RDFD's will also relate to additional subclasses of FIFO Petri Nets. Therefore, it would be reasonable to join research efforts on FDFD's and on FIFO Petri Nets.

So far, there remain several open decidability problems for subclasses of RDFD's, since the related problem is open for the corresponding subclass of FIFO Petri Nets. It is conjectured ([FR88]) that most of these problems are decidable even though no proof or algorithm exists at this time. Further work has to be done to identify which problem is (or is not) decidable for which subclass of RDFD's/FIFO Petri Nets. Another interesting approach would be the extension of a method well-known for Petri Nets — the reduction of the number of places of the Petri Net (e. g., [BR76]) — towards subclasses of RDFD's/FIFO Petri Nets with the intent to solve decidability questions more efficiently.

Finally, we must admit that there was no effort made so far that deals with the complexity and efficiency of the decidability algorithms. Even though many problems have been identified as decidable

for particular subclasses of RDFD's, no efficient algorithm has yet been given. It is desireable to determine (lower and upper) bounds for (time and space) complexity of possible algorithms and evaluate given current (and future) algorithms with respect to these bounds.

Acknowledgements

Symanzik's research was partially supported by a German "DAAD–Doktorandenstipendium aus Mitteln des zweiten Hochschulsonderprogramms".

5 TIMED DATA FLOW DIAGRAMS

A conference contribution submitted to Ada-Europe '97, London, UK (June 1997)

Jürgen Symanzik[1] and Albert L. Baker[2]

Abstract

Data Flow Diagrams (DFD's) are widely used in industry to express requirements specifications. However, as used in practice, there has been no precise semantics for DFD's, let alone an incorporation of a model of time. In this paper, we augment the Formalized Data Flow Diagrams (FDFD's) defined in [LWBL96] by adding a deterministic (or stochastic) time behavior for the consumption of values from in-flows to processes and the production of values to the out-flows from processes. We call our new FDFD model Timed (or Stochastic) Data Flow Diagrams (TDFD's or SDFD's). We identify two factors in determining how time can affect the choice of how an FDFD can change state. The first factor has to do with when the decision is made as to which state transition will be next occur. The two possibilities are a *Preselection Policy* and a *Race Policy*. The other timing factor is the past history of an FDFD execution. We identify three alternatives: *Resampling*, *Work Age Memory*, and *Enabling Age Memory*. Certain combinations of these alternatives allow us to model systems where components are competing for limited resources. Other combinations allow us to model systems where components work concurrently. Preemption can also be modeled using these alternatives. Major results for the quantitative and qualitative analysis of TDFD's can be borrowed from the literature on Timed Petri Nets.

Keywords

Statistical Software Engineering, Formal Methods, Concurrent and Distributed Systems, Software Specification, Formalized Data Flow Diagrams, Timed Petri Nets.

[1] Primary researcher and author.
[2] Professor, Department of Computer Science, Iowa State University.

5.1 Introduction

If formal specification methods, at an appropriate level of abstraction, could support

- reasonable and convenient modeling of system timing behavior and
- direct specification execution

their use would be far more prevalent. The results presented in this paper provide a formalization of an already widely-used specification method that supports modeling of the timing behavior of concurrent and distributed systems and that can be directly executed. Our approach is to integrate models of timing behavior and semantic rigor with traditional Data Flow Diagrams (DFD's).

DFD's are the basis of the software development methodology known as "Structured Analysis" (SA) ([DeM78], [WM85a]). DFD's are popular because their graphical representation and hierarchical structure allow some comprehension by users with non-technical backgrounds and they can also serve as an initial characterization of software architecture.

The primary components of DFD's are bubbles and flows. In the graphical representation, bubbles are drawn as circles while flows are drawn as arcs connecting the bubbles. Hence, formally, a DFD is a directed bipartite graph. Within this model, bubbles can represent processes in a distributed or concurrent system. Flows can then represent message paths. A bubble consumes the information (values) on its in-flows, and produces information on its out-flows.

Numerous formalizations of DFD's have appeared in the technical literature, e. g., in [DeM78], [WM85a], [WM85b], [Har87], [TP89], [You89], [Har92], and [Har96]. In this paper we use the definitions of Formalized Data Flow Diagrams (FDFD's) developed by Coleman, Wahls, Baker, and Leavens in [Col91], [CB94], [WBL93], and [LWBL96]. [War86] introduces a transformation schema that allows to represent the control and timing aspects of a real system modeled as a DFD. However, this approach has very little in common with computational models such as Timed Petri Nets (TPN's) or the concept of time in FDFD's, introduced in this paper, where time is used to describe the behavior and to analyze quantitative properties of the real system. In particular, for the definitions and example in this paper, we use the notation from [LWBL96]. The potential for direct execution of these FDFD's is presented in [WBL94].

It has been shown recently that a subclass of FDFD's, so called persistent flow-free Reduced Data Flow Diagrams (PFF-RDFD's) is Turing equivalent ([SB96a]). On the other hand, features such as persistent flows, stores, and the facility to test for empty flows that are widely used in applications of

FDFD's, only add to the expressive convenience of FDFD's, and do not raise the power of the model beyond that of Turing Machines ([SB96b]).

To-date, and to the best of our knowledge, there does not exist any extension of DFD's that includes the notion of time. We have augmented FDFD's to include timing, and refer to such DFD's as Timed Data Flow Diagrams (TDFD's) or Stochastic Data Flow Diagrams (SDFD's) acknowledging the stochastic models we adopt for time. Because of the wide use of DFD's for requirements specifications, we completely maintain the original syntax and semantics of FDFD's and, in what we hope is both an intuitive and general manner, add a model of time to the operational semantics of FDFD's.

We have borrowed from the work on Timed Petri Nets (TPN's), in particular from [MBB+85], for incorporating a model of time into FDFD's. However, TPN's are used primarily to capture the requisite synchronization in concurrent and distributed systems, but do not usually represent the full functional behavior of the systems. Thus, if one is developing a client server system with replicated servers, TPN's can be used to indicate the synchronization of communication between servers in satisfying a client request, but would not capture the particulars of the data interchanged between servers, nor the actual responses to clients. By relating the existing analytical results for TPN's to our model of time in TDFD's, people who currently use DFD's as a specification technique can immediately use the more powerful timed model and achieve the same type of results for issues like deadlock and race conditions available for analogous TPN's.

A reasonable approach to the modeling of time in FDFD's is to define a stochastic time behavior for the consumption of in-flow items as well as a stochastic time behavior for the production of items on the out-flow. In a different approach, we could assign message passing times to the FDFD. Other models for times, or mixtures of several approaches, might be adopted. In this paper we only use the first approach. But it is worth noting that we have found our model of time to be expressively convenient and that the particular choice may not be of theoretical importance. It is established in [BR90] that, on a fundamental level, any type of TPN is sufficient, as long as it contains nonzero delays. We elsewhere ([SB96a], [SB96c]) argue the tight relationship between (subclasses of) FDFD's, Petri Nets (e. g., [Pet81]), and FIFO Petri Nets (introduced in [MM81]).

The work presented here covers one of the interrelated aspects of Statistics and Software Engineering, combined in the new interdisciplinary field "Statistical Software Engineering"[3]. The introduction of timing and the related statistical analysis will allow as early as in the Specifications phase of the spiral

[3]Statistical Software Engineering: "The interdisciplinary field of statistics and software engineering specializing in the use of statistical methods for controlling and improving the quality and productivity of the practices used in creating software." ([Nat96], p. 5)

software development process model ([Nat96], p. 63) to decide whether quantitative requirements of the software system are fulfilled.

In Section 5.2 of this paper, we will summarize basic definitions for FDFD's. Timed (Stochastic) Data Flow Diagrams will be introduced in Section 5.3. In Section 5.4, we describe a Producer/Consumer Model as a TDFD and consider possible execution policies. We conclude this paper with an overview on future work in Section 5.5.

5.2 Definitions

The definition of FDFD's from [LWBL96] is:

Definition (5.2.1): A *Formalized Data Flow Diagram* (FDFD) is a quintuple

$$FDFD = (B, FLOWNAMES, TYPES, P, F),$$

where B is a set of *bubbles*. $FLOWNAMES$ is a set of *flows*, $TYPES$ is a set of *types*, P is the set $\{persistent, consumable\}$ and $F = B \times FLOWNAMES \times TYPES \times B \times P$. The following notational convention for members from these domains is used: $b \in B, fn \in FLOWNAMES, T \in TYPES, p \in P, f \in F$. ■

While a more rigorous definition of the mappings used to define the execution behavior of FDFD's is contained in [LWBL96], a less formal explanation follows.

Consumable flows are modeled as infinite queues of values and persistent flows are modeled as shared variables for which the source bubble can write the value and the destination bubble can read the value. Intuitively, the semantics of FDFD's is based on a two-step firing rule for bubbles. Each bubble b is either in one of two modes: *idle* or *working*. The state of an FDFD is the current value on all the flows and the current mode of each bubble (along with what values were consumed by bubbles in the *working* mode, at the point they went from *idle* to *working*).

If bubble b is *idle*, then its change of state to *working* is predicated by an enabling rule, *Enabled* which is just an assertion over the values on its in-flows. If *Enabled* is satisfied for bubble b and the values of the in-flows to b, then b is a candidate for firing. If b is selected for firing, then the values on the in-flows to b which satisfied the enabling rule are consumed (or copied from persistent flows), and b enters *working* mode.

If a bubble b is *working*, it is also a candidate for firing. When b is selected for execution, the values that were previously consumed (or copied) are used in a postcondition assertion that defines the values to be output from b on out-flows from b.

From this basic definition of FDFD's, we can define a sequence of firing steps in the execution of an FDFD:

Definition (5.2.2): A *firing sequence* (*computation sequence*) of an FDFD is a possibly infinite sequence $(b_i, a_i, j_i) \in B \times \{C, P\} \times \mathbb{N}, i \geq 0$, such that, if transition (b_i, a_i, j_i) is fired in state (bm, r, fs), then

$$(fs', r') = \begin{cases} (Consume(b_i))_{j_i}(fs, r), & \text{if } a_i = C \\ (Produce(b_i))_{j_i}(fs, r). & \text{if } a_i = P \end{cases}$$

$$bm'(b_i) = \begin{cases} working, & \text{if } a_i = C \\ idle, & \text{if } a_i = P \end{cases}$$

$$bm'(b) = bm(b) \quad \forall b \in B - \{b_i\}$$

and

$$(bm, r, fs) \rightarrow (bm', r', fs').$$

We introduce the notation $(bm, r, fs)[(b, a, j)]$ to indicate that transition (b, a, j) is fireable in state (bm, r, fs) and $(bm, r, fs)[(b, a, j)](bm', r', fs')$ to indicate that state (bm', r', fs') is reached upon the firing of transition (b, a, j) in state (bm, r, fs).

By induction, we extend this notation for firing sequences:

$$(bm_0, r_0, fs_0)[(b_1, a_1, j_1), \ldots, (b_{n-1}, a_{n-1}, j_{n-1}), (b_n, a_n, j_n)]$$

is used to indicate that transition (b_n, a_n, j_n) is fireable in state $(bm_{n-1}, r_{n-1}, fs_{n-1})$, given that

$$(bm_0, r_0, fs_0)[(b_1, a_1, j_1), \ldots, (b_{n-1}, a_{n-1}, j_{n-1})](bm_{n-1}, r_{n-1}, fs_{n-1})$$

holds. By analogy, we use

$$(bm_0, r_0, fs_0)[(b_1, a_1, j_1), \ldots, (b_n, a_n, j_n)](bm_n, r_n, fs_n)$$

to indicate that state (bm_n, r_n, fs_n) is reached upon the firing of the sequence $(b_1, a_1, j_1), \ldots, (b_n, a_n, j_n)$.

■

Definition (5.2.3): The *set of firing sequences* (*set of computation sequences, language*) of an FDFD, denoted by $FS(FDFD, \gamma_{initial})$, is the set containing all firing sequences that are possible for

this FDFD, given $\gamma_{initial} = (bm_{initial}, r_{initial}, fs_{initial})$, i. e.,

$$FS(FDFD, \gamma_{initial}) = \{s \mid s \in (B \times \{C, P\} \times I\!N)^* \wedge \gamma_{initial}[s]\}. \quad \blacksquare$$

Definition (5.2.4): The *Reachability Set* of an FDFD, denoted by $RS(FDFD, \gamma_{initial})$, is the set of states $\gamma = (bm, r, fs)$ that are reachable from $\gamma_{initial} = (bm_{initial}, r_{initial}, fs_{initial})$, i. e.,

$$RS(FDFD, \gamma_{initial}) = \{\gamma \mid \gamma \in \Gamma \wedge \exists s \in FS(FDFD, \gamma_{initial}) : \gamma_{initial}[s]\gamma\}. \quad \blacksquare$$

Definition (5.2.5): Let $EN(\gamma) \subseteq B \times \{C, P\} \times I\!N$ be the set of transitions that are enabled in state $\gamma = (bm, r, fs)$, i. e..

$$EN(\gamma) = \{s \mid s \in (B \times \{C, P\} \times I\!N) \wedge \gamma[s]\}. \quad \blacksquare$$

5.3 Stochastic Data Flow Diagrams

We will make use of random variables to specify the time behavior of FDFD's. Therefore, Stochastic Data Flow Diagram (SDFD) is a more appropriate name for our new model. We try to follow the general approach of Stochastic Petri Nets given in [MBB+85] when defining SDFD's, when considering the impact of different execution policies on the semantics of the model, and when allowing a general time distribution that induces an associated stochastic process.

We start to describe the behavior of a TDFD by describing a possible timed firing sequence of an FDFD.

Definition (5.3.1): A *timed firing sequence* (TFS) of an FDFD with initial state $\gamma_{initial}$ is a pair $tfs = (s, \tau)$, where $s \in FS(FDFD, \gamma_{initial})$ and τ is a non-decreasing sequence (of the same length) of real non-negative values representing the instants of firing (called *epochs*) of each transition, such that consecutive transitions (b_i, a_i, j_i) and $(b_{i+1}, a_{i+1}, j_{i+1})$ correspond to ordered epochs $\tau_i \leq \tau_{i+1}$. The time intervals $[\tau_i, \tau_{i+1})$ between consecutive epochs represent the periods in which the FDFD remains in state γ_i (assuming $\tau_0 = 0$). A *history* of the FDFD up to the kth epoch τ_k is denoted by $Z(k)$. $\blacksquare$

The introduction of a stochastic time behavior for FDFD's will allow us to describe (in a probabilistic sense) the future behavior of a system from the knowledge of the past history and the current state.

Definition (5.3.2): Let $\underline{Z} = Z(k)$ be a history of the FDFD up to (and including) the kth epoch, and $\underline{\gamma} = \gamma_k$ be the state entered by firing transition (b_k, a_k, j_k). We assume that for all k, $\underline{Z}$, and $\underline{\gamma}$, the following joint distribution functions can be uniquely determined:

$$F_{\Gamma,X}((b,a,j),x \mid \underline{\gamma},\underline{Z}) = Pr(\Gamma = (b,a,j), X \leq x \mid \underline{\gamma},\underline{Z}) \qquad (5.3.2.1)$$

■

The distribution above depends on two random variables: The discrete random variable Γ represents the transition that will fire. The sample space for Γ is the set of transitions enabled in $\underline{\gamma}$, i. e., $\Omega_\Gamma = EN(\underline{\gamma})$. The continuous random variable X represents the time that elapses from entering $\underline{\gamma}$ up to the next transition epoch, i. e., the time interval $\tau_{k+1} - \tau_k$. The sample space for X are positive real numbers including 0 (same time as previous transition) and ∞ (never), i. e., $\Omega_X = \mathbb{R}^+_\infty$.

Note that $\underline{\gamma}$ is known from $\underline{Z}$, but we have explicitly indicated the dependence on $\underline{\gamma}$ since quite often. $\underline{\gamma}$ is the only component that influences the joint distribution functions (5.3.2.1). These distributions must be defined for all transitions $(b,a,j) \in EN(\underline{\gamma})$.

We define the marginal probability function of selecting (b,a,j) to be the next transition to fire as

$$\begin{aligned} p_{(b,a,j)}(\underline{\gamma},\underline{Z}) &= Pr(\Gamma = (b,a,j) \mid \underline{\gamma},\underline{Z}) \\ &= \lim_{t\to\infty} F_{\Gamma,X}((b,a,j),t \mid \underline{\gamma},\underline{Z}) \\ &= \int_0^\infty d_x F_{\Gamma,X}((b,a,j),x \mid \underline{\gamma},\underline{Z}) \end{aligned} \qquad (5.3.2.2)$$

and the marginal distribution function of the time spent in state $\underline{\gamma}$ before the next epoch is reached as

$$F_X(x \mid \underline{\gamma},\underline{Z}) = \sum_{s\in EN(\underline{\gamma})} F_{\Gamma,X}(s,x \mid \underline{\gamma},\underline{Z}). \qquad (5.3.2.3)$$

Definition (5.3.3): A *Stochastic Data Flow Diagram* (SDFD) is an FDFD (with initial state $\gamma_{initial}$) with a set of specifications for calculating the joint distribution functions $F_{\Gamma,X}((b,a,j),x \mid \underline{\gamma},\underline{Z})$ for all $\underline{\gamma}$ and $\underline{Z}$, and with an initial probability distribution on the Reachability Set $RS(FDFD,\gamma_{initial})$.

■

Due to this definition, the ensemble of possible executions of a SDFD, together with the probability measure induced on it by assigning the firing distributions $F_{\Gamma,X}((b,a,j),x \mid \underline{\gamma},\underline{Z})$, describes a stochastic process with a discrete state space isomorphic to a subset of $RS(FDFD,\gamma_{initial})$ of the associated FDFD. For the initial probability distribution on $RS(FDFD,\gamma_{initial})$, we assume that $Pr(\text{system is in state } \gamma_{initial} \text{ at time } \tau_0 = 0) = 1$.

When in a given state $\underline{\gamma}$ only one transition is enabled, say (b, a, j), the calculation of

$$F_{\Gamma,X}((b,a,j),x \mid \underline{\gamma},\underline{Z})$$

requires only to determine the distribution of the time spent in $\underline{\gamma}$, possibly conditioned on the past history $\underline{Z}$.

When a state $\underline{\gamma}$ enables at least two transitions, the computation of the distributions

$$F_{\Gamma,X}((b,a,j),x \mid \underline{\gamma},\underline{Z})$$

requires the knowledge of the policy used for the selection of the transition that fires. We consider two possible policies:

Definition (5.3.4): The *Preselection Policy*: When the SDFD enters state $\underline{\gamma}$, a transition s is selected among those in $EN(\underline{\gamma})$ according to its probability $p_s(\underline{\gamma},\underline{Z})$, Then, s will fire after a random delay with distribution $F_{X|\Gamma}(x \mid s,\underline{\gamma},\underline{Z})$. Thus, the selection of the transition that actually fires does not depend upon the associated delay. Otherwise, once a transition has been selected, the sojurn time in $\underline{\gamma}$ does not depend upon the delays associated to the other transitions. Therefore, a model with preselection policy requires the specification of both the probabilities $p_s(\underline{\gamma},\underline{Z})$ and the conditional distributions $F_{X|\Gamma}(x \mid s,\underline{\gamma},\underline{Z})$.

Equation (5.3.2.1) can be rewritten as

$$F_{\Gamma,X}(s,x \mid \underline{\gamma},\underline{Z}) = p_s(\underline{\gamma},\underline{Z}) \cdot F_{X|\Gamma}(x \mid s,\underline{\gamma},\underline{Z}) \qquad (5.3.4.1)$$

where $F_{X|\Gamma}$ represents the conditional distribution of the firing delays conditioned on $\underline{\gamma},\underline{Z}$, and the fact that s is the transition that will actually fire. Obviously, $\sum_{s_i \in EN(\underline{\gamma})} p_{s_i}(\underline{\gamma},\underline{Z}) = 1$. ■

Definition (5.3.5): The *Race Policy*: When the SDFD enters state $\underline{\gamma}$, for each transition $s_i \in EN(\underline{\gamma})$ a random sample ϑ_i from θ_i is extracted from the joint distribution

$$\Phi_{\theta_1,\ldots,\theta_{|EN(\underline{\gamma})|}}(x_1,\ldots,x_{|EN(\underline{\gamma})|} \mid \underline{\gamma},\underline{Z}) = Pr(\theta_1 \le x_1,\ldots,\theta_{|EN(\underline{\gamma})|} \le x_{|EN(\underline{\gamma})|} \mid \underline{\gamma},\underline{Z}) \qquad (5.3.5.1)$$

The minimum ϑ_i of the samples $\vartheta_1,\ldots,\vartheta_{|EN(\underline{\gamma})|}$ determines two properties: the transition s_i which will actually fire and the sojorn time ϑ_i in $\underline{\gamma}$. We consider only the case where all random variables θ_i are stochastically independent. Then, (5.3.5.1) is uniquely determined by the marginal distributions

$$\phi_i(x \mid \underline{\gamma},\underline{Z}) = Pr(\theta_i \le x \mid \underline{\gamma},\underline{Z}), i = 1,\ldots,\mid EN(\underline{\gamma}) \mid. \qquad (5.3.5.2)$$

Now, the joint distribution function (5.3.2.1) can be expressed as

$$F_{\Gamma,X}(s_i, x \mid \underline{\gamma}, \underline{Z}) = \int_0^x \left(\prod_{\substack{j=1,\ldots,|EN(\underline{\gamma})| \\ j \neq i}} [1 - \phi_j(u \mid \underline{\gamma}, \underline{Z})] \right) d_u \phi_i(u \mid \underline{\gamma}, \underline{Z}). \qquad (5.3.5.3)$$

Therefore, the marginal probability function (5.3.2.2) can be rewritten as

$$\begin{aligned} p_{s_i}(\underline{\gamma}, \underline{Z}) &= \lim_{t \to \infty} F_{\Gamma,X}(s_i, t \mid \underline{\gamma}, \underline{Z}) \\ &= \int_0^\infty \left(\prod_{\substack{j=1,\ldots,|EN(\underline{\gamma})| \\ j \neq i}} [1 - \phi_j(u \mid \underline{\gamma}, \underline{Z})] \right) d_u \phi_i(u \mid \underline{\gamma}, \underline{Z}) \qquad (5.3.5.4) \end{aligned}$$

and the marginal distribution function (5.3.2.3) can be expressed as

$$\begin{aligned} F_X(x \mid \underline{\gamma}, \underline{Z}) &= 1 - \prod_{j=1,\ldots,|EN(\underline{\gamma})|} Pr(\theta_j > x \mid \underline{\gamma}, \underline{Z}) \\ &= 1 - \prod_{j=1,\ldots,|EN(\underline{\gamma})|} [1 - \phi_j(x \mid \underline{\gamma}, \underline{Z})]. \qquad (5.3.5.5) \end{aligned}$$

Thus, a model with race policy requires the specification of the marginal distributions $\phi_i(x \mid \underline{\gamma}, \underline{Z}), i = 1, \ldots, |EN(\underline{\gamma})|$, only. ■

Now, we consider three possible ways to deal with the past history $\underline{Z}$:

Resampling: The distributions $F_{\Gamma,X}((b,a,j), x \mid \underline{\gamma}, \underline{Z})$ are independent of $\underline{Z}$, but they may depend on the current state $\underline{\gamma}$.

Work Age Memory: The distributions $F_{\Gamma,X}((b,a,j), x \mid \underline{\gamma}, \underline{Z})$ depend on the past history $\underline{Z}$ through a new variable, a so–called work age variable, associated with each transition (b,a,j). The work age variables accumulate the work done — for each transition from its last firing up to the considered epoch. The distributions $F_{\Gamma,X}((b,a,j), x \mid \underline{\gamma}, \underline{Z})$ represent the residual times needed for the transitions to complete.

Enabling Age Memory: The distributions $F_{\Gamma,X}((b,a,j), x \mid \underline{\gamma}, \underline{Z})$ depend on the past history $\underline{Z}$ through a new variable, a so–called enabling age variable, associated with each transition (b,a,j). The enabling age variables accumulate the work done — for each transition from the last instant at which it has become enabled up to the considered epoch. The distributions $F_{\Gamma,X}((b,a,j), x \mid \underline{\gamma}, \underline{Z})$ represent the residual times needed for the transitions to complete.

Combining the two policies for selecting a transition that fires with the three methods of dealing with the past history $\underline{Z}$ results in six different execution policies. Four of them are intuitively easy to understand while two of them are more complex and can be neglected for practical purposes.

Race Policy with Resampling (RR): This execution policy best describes the behavior of a set of parallel competing (conflicting) transitions. The first transition to terminate determinates a change in the system state. The work done by all other transitions that do not complete is lost. Therefore, this policy seems to be interesting only in the case of conflicting transitions that make use of the same resources. Note that this only happens for FDFD's if for a given input that has been read a bubble can nondeterministically select among two or more alternatives which output to produce when changing its mode from *working* to *idle*.

Race Policy with Work Age Memory (RW): Here, simultaneous transitions are described. The first transition to terminate determines a change in the system state. However, the work done by all other transitions that do not complete is not lost. Instead, it is assumed that all the work done by each transition is being accumulated from when it is first enabled up to the current firing. After this firing, a transition will resume its work in the next state that enables it, from the point at which it has been interrupted. Therefore, this execution policy is useful in situations where conflicting and concurrent transitions can happen.

Race Policy with Enabling Age Memory (RE): Once again, simultaneous transitions are described. The first transition to terminate determines a change in the system state. However, in this case, the work done by all other transitions that do not complete is lost, unless they remain enabled in the new state that is reached through the current firing. Therefore, this execution policy is useful in situations where conflicting and concurrent transitions can happen, but it also allows preemption.

Preselection with Resampling (PR): This execution policy can be used when a set of conflicting transitions can not perform in parallel. Before the system can enter a new state, it has to choose which of the conflicting transitions will fire next. Once choosen, this transition will perform until completion and the system will enter the new state.

Preselection with Work Age Memory (PW): In this execution policy, the transition that will determine the stage change is choosen immediately when a new state has been reached. Then, this transition performs until completion and the system will enter the next state. All other transitions that were enabled but have not been choosen execute in parallel to the choosen one until the state change caused by the choosen transition occurs. Each transition will resume work in the next state where it is enabled, continuing from the point that has been reached when the state change occurred. Of course, this may cause some paradoxon: It may happen that the

choosen transition performs longer than some of the transitions that have not been choosen and have terminated without being allowed to induce a state change.

Preselection with Enabling Age Memory (PE): Similarly, the transition that will determine the stage change is choosen immediately when a new state has been reached. Then, this transition performs until completion and the system will enter the next state. All other transitions that were enabled but have not been choosen execute in parallel to the choosen one until the state change caused by the choosen transition occurs. However, if a transition is not enabled in the new state that is reached through the current firing, all its accumulated work is lost. The same paradoxon as in the previous case may occur.

5.4 Example of a Producer/Consumer Model

The example in this section shows a Producer/Consumer model where the producer can work whenever it is ready, while the consumer has to wait for inputs from the producer. The given distributions do not really provide a useful application, but have been choosen to demonstrate the different behavior of the model under different execution policies. In fact, for most of the policies, the producer will produce items at a faster rate then the consumer can consume, resulting in a (permanently) increasing queue length of flow *f*.

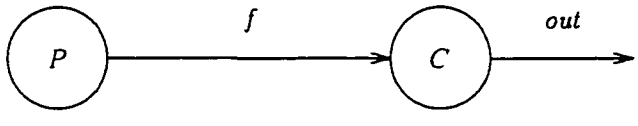

Figure 5.1: Example of a SDFD.

The mappings *Enabled*, *Consume*, and *Produce* for the SDFD shown in Figure 5.1 are specified as follows:

$$Enabled(P) = \lambda fs\ .\ true$$

$$Enabled(C) = \lambda fs\ .\ (\neg IsEmpty(f) \wedge Head(fs(f)) = 0) \vee (\neg IsEmpty(f) \wedge Head(fs(f)) = 1)$$

	s	t_s
$Consume(P) = \lambda(fs, r)\ .\ \{(fs, r)\}$	(P, C, 1)	2
$Consume(C) = \lambda(fs, r)\ .$		

$\{$**if** $(\neg IsEmpty(f) \wedge Head(fs(f)) = 0)$

then $In(f, C)(fs, r)$ (C, C, 1) 4

fi,

if $(\neg IsEmpty(f) \wedge Head(fs(f)) = 1)$

then $In(f, C)(fs, r)$ (C, C, 2) 5

fi

$\}$

$Produce(P) = \lambda(fs, r)$.

$\{Out(0, f, P)(fs, r)$, (P, P, 1) 2

$Out(1, f, P)(fs. r)\}$ (P, P, 2) 3

$Produce(C) = \lambda(fs, r)$.

$\{$**if** $r(C)(f) = 0$

then $Out(a, out, C)(fs, r)$ (C, P, 1) 3

$\square\, Out(b, out, C)(fs, r)$ (C, P, 2) 4

fi,

if $r(C)(f) = 1$

then $Out(c. out. C)(fs, r)$ (C, P, 3) 6

fi

$\}$

Now, we consider the effect of different execution policies on this basic model. The symbol $\surd$ marks the transition that actually fires. With "Acc" we denote the time accumulated for a transition that did not fire.

Race Policy:

We define the following marginal distributions $\phi_i(x \mid \underline{\gamma}. \underline{Z})$ for the race policy:

$$Pr(\theta_s = t_s \mid \underline{\gamma}. \underline{Z}) = 1 \ \forall s \in EN(\underline{\gamma}) \ \forall \underline{\gamma} \ \forall \underline{Z}$$

All these distributions are Dirac distributions. From the statistical point of view, they are degenerate distributions where all the mass is assigned to the point t_s. From the applied point of view, these distributions are used to express a deterministic time for each transition.

Race/Resampling

There is only one possible timed firing sequence $tfs = (s, \tau)$ for this execution policy:

i	0	1	2	3	4
τ	0	2	4	6	8
s		(P, C, 1)	(P, P, 1)	(P, C, 1)	(P, P, 1)
$EN(\underline{\gamma})$	(P, C, 1)$\surd$	(P, P, 1)$\surd$	(P, C, 1)$\surd$	(P, P, 1)$\surd$	(P, C, 1)
		(P, P, 2)	(C, C, 1)	(P, P, 2)	(C, C, 1)
				(C, C, 1)	

- $\tau_1 = 2$: (P, C, 1) is the only possible transition and executes at time 2.
- $\tau_2 = 4$: (P, P, 1) and (P, P, 2) compete and will terminate at time 4 and 5, respectively. Therefore, (P, P, 1) executes at time 4.
- $\tau_3 = 6$: (P, C, 1) and (C, C, 1) compete and will terminate at time 6 and 8, respectively. Therefore, (P, C, 1) executes at time 6.
- $\tau_4 = 8$: (P, P, 1), (P, P, 2), and (C, C, 1) compete and will terminate at time 8, 9, and 10, respectively. Therefore, (P, P, 1) executes at time 8.
- $\tau_5 = 10$: In analogy to τ_3.

Race/Work Age

We indicate the timed firing sequence $tfs = (s, \tau)$ for the first 5 epochs:

i	0	1	2	3	4	5
τ	0	2	4	6	7	8
s		(P, C, 1)	(P, P, 1)	(P, C, 1)	(P, P, 2)	(C, C, 1)
$EN(\underline{\gamma})$/Acc	(P, C, 1)/0 $\surd$	(P, P, 1)/0 $\surd$	(P, C, 1)/0 $\surd$	(P, P, 1)/0	(P, C, 1)/0	(P, C, 1)/1
		(P, P, 2)/0	(C, C, 1)/0	(P, P, 2)/2 $\surd$	(C, C, 1)/3 $\surd$	(C, P, 1)/0
				(C, C, 1)/2		(C, P, 2)/0
$\neg EN(\underline{\gamma})$/Acc			(P, P, 2)/2		(P, P, 1)/1	(P, P, 1)/1

- $\tau_1 = 2$: (P, C, 1) is the only possible transition and executes at time 2.
- $\tau_2 = 4$: (P, P, 1) and (P, P, 2) compete and will terminate at time 4 and 5, respectively. Therefore, (P, P, 1) executes at time 4. (P, P, 2) accumulates 2 time units of work, but is not enabled past $\tau_2 = 4$.
- $\tau_3 = 6$: (P, C, 1) and (C, C, 1) compete and will terminate at time 6 and 8, respectively. Therefore, (P, C, 1) executes at time 6. (C, C, 1) accumulates 2 time units of work and is enabled past $\tau_3 = 6$. (P, P, 2) is enabled again past $\tau_3 = 6$.

- $\tau_4 = 7$: (P, P, 1), (P, P, 2), and (C, C, 1) compete and will terminate at time 8, 7, and 8, respectively. Therefore, (P, P, 2) executes at time 7. (P, P, 1) accumulates 1 time unit of work, but is not enabled past $\tau_4 = 7$. (C, C, 1) accumulates 1 time unit of work and is enabled past $\tau_4 = 7$.

- $\tau_5 = 8$: (P, C, 1) and (C, C, 1) compete and will terminate at time 9 and 8, respectively. Therefore, (C, C, 1) executes at time 8. (P, C, 1) accumulates 1 time unit of work and is enabled past $\tau_5 = 8$. (P, P, 1) remains disabled past $\tau_5 = 8$.

Race/Enabling Age

We indicate the timed firing sequence $tfs = (s, \tau)$ for the first 5 epochs:

i	0	1	2	3	4	5
τ	0	2	4	6	8	10
s		(P, C, 1)	(P, P, 1)	(P, C, 1)	(P, P, 1)	(P, C, 1)
					(C, C, 1)	
$EN(\gamma)$/Acc	(P, C, 1)/0 $\surd$	(P, P, 1)/0 $\surd$	(P, C, 1)/0 $\surd$	(P, P, 1)/0 $\surd$	(P, C, 1)/0 $\surd$	(P, P, 1)/0
		(P, P, 2)/0	(C, C, 1)/0	(P, P, 2)/0	(C, P, 1)/0	(P, P, 2)/0
				(C, C, 1)/2 $\surd$	(C, P, 2)/0	(C, P, 1)/2
						(C, P, 2)/2

- $\tau_1 = 2$: (P, C, 1) is the only possible transition and executes at time 2.

- $\tau_2 = 4$: (P, P, 1) and (P, P, 2) compete and will terminate at time 4 and 5, respectively. Therefore, (P, P, 1) executes at time 4. (P, P, 2) looses the work accumulated so far since it is not enabled past $\tau_2 = 4$.

- $\tau_3 = 6$: (P, C, 1) and (C, C, 1) compete and will terminate at time 6 and 8, respectively. Therefore, (P, C, 1) executes at time 6. (C, C, 1) accumulates 2 time units of work and is enabled past $\tau_3 = 6$.

- $\tau_4 = 8$: (P, P, 1), (P, P, 2), and (C, C, 1) compete and will terminate at time 8, 9, and 8, respectively. We have to consider two cases:

  - (P, P, 1) executes at time 8, then (C, C, 1) executes at time 8.

  - (C, C, 1) executes at time 8, then (P, P, 1) executes at time 8.

  Since both transitions depend on different resources and the execution of one of them does not disable the other one, the results are the same. (P, P, 2) looses the work accumulated so far since it is not enabled past $\tau_4 = 8$.

- $\tau_5 = 10$: (P, C, 1), (C, P, 1), and (C, P, 2) compete and will terminate at time 10, 11, and 12, respectively. Therefore, (P, C, 1) executes at time 10. (C, P, 1) accumulates 2 time units of work and is enabled past $\tau_5 = 10$. (C, P, 2) accumulates 2 time units of work and is enabled past $\tau_5 = 10$.

Preselection Policy:

We consider three different cases of the Preselection/Resampling policy.

- First, we define

$$p_s(\underline{\gamma}, \underline{Z}) = \frac{1}{\mid EN(\underline{\gamma}) \mid} \ \forall s \in EN(\underline{\gamma}) \ \forall \underline{\gamma} \ \forall \underline{Z}$$

and

$$Pr(X = t_s \mid s, \underline{\gamma}, \underline{Z}) = 1 \ \forall s \in EN(\underline{\gamma}) \ \forall \underline{\gamma} \ \forall \underline{Z}.$$

Here are all possible timed firing sequences $tfs = (s, \tau)$ according to this policy and distribution for the first 4 epochs:

i	0	1	2	3	4
$s : \tau$		2 : (P, C, 1)	4 : (P, P, 1)	6 : (P, C, 1)	8 : (P, P, 1)
					9 : (P, P, 2)
					10 : (C, C, 1)
				8 : (C, C, 1)	10 : (P, C, 1)
					11 : (C, P, 1)
					12 : (C, P, 2)
			5 : (P, P, 2)	7 : (P, C, 1)	9 : (P, P, 1)
					10 : (P, P, 2)
					12 : (C, C, 2)
				10 : (C, C, 2)	12 : (P, C, 1)
					16 : (C, P, 3)

- Now, we change the selection probabilities to

$$p_{(P,C,1)}(\underline{\gamma}, \underline{Z}) = 1 \ \forall \underline{\gamma} : (idle(P)) \ \forall \underline{Z}$$

$$p_{(P,P,1)}(\underline{\gamma}, \underline{Z}) = 1 \ \forall \underline{\gamma} : (working(P)) \ \forall \underline{Z}$$

and

$$p_s(\underline{\gamma}, \underline{Z}) = 0 \text{ for all other } s \in EN(\underline{\gamma}) \ \forall \underline{Z}$$

but maintain the delay

$$Pr(X = t_s \mid s, \underline{\gamma}, \underline{Z}) = 1 \ \forall s \in EN(\underline{\gamma}) \ \forall \underline{\gamma} \ \forall \underline{Z}.$$

We make use of 0–probabilities to disable some transitions that are possible firing candidates according to $EN(\underline{\gamma})$. Actually, we only allow bubble P to proceed. This yields exactly the same timed firing sequence as for the Race/Resampling policy.

- Finally, we consider a more realistic model by choosing the following selection probabilities

$$p_{(P,C,1)}(\underline{\gamma}, \underline{Z}) = 1 \ \forall \underline{\gamma} : (idle(P) \wedge idle(C) \wedge IsEmpty(f)) \ \forall \underline{Z}$$

$$p_{(P,P,1)}(\underline{\gamma}, \underline{Z}) = 1 \ \forall \underline{\gamma} : (working(P)) \ \forall \underline{Z}$$

$$p_{(C,C,1)}(\underline{\gamma}, \underline{Z}) = 1 \ \forall \underline{\gamma} : (idle(P) \wedge idle(C) \wedge \neg IsEmpty(f)) \ \forall \underline{Z}$$

$$p_{(C,P,1)}(\underline{\gamma}, \underline{Z}) = 1 \ \forall \underline{\gamma} : (working(C)) \ \forall \underline{Z}$$

and

$$p_s(\underline{\gamma}, \underline{Z}) = 0 \text{ for all other } s \in EN(\underline{\gamma}) \ \forall \underline{Z}$$

but maintain the delay

$$Pr(X = t_s \mid s, \underline{\gamma}, \underline{Z}) = 1 \ \forall s \in EN(\underline{\gamma}) \ \forall \underline{\gamma} \ \forall \underline{Z}.$$

Once again, we make use of 0–probabilities to disable some transitions. The effect of this policy and distribution is a timed firing sequence where (P, C, 1), (P, P, 1), (C, C, 1), and (C, P, 1) alternate in this order.

5.5 Future Directions

One of the first things to be done in the future is an overview of the types of stochastic processes associated to our TDFD, similar to the characterization of the stochastic process that is underlying a Stochastic Petri Net ([CGL94]). Different types of probability distributions and execution policies will result in stochastic processes of different flavors, some of them easy to analyze and some of them difficult to capture. For Timed Petri Nets (TPN's), most work has been done for Exponential distributions (e. g., [MC87]), associated to a Markov process which is usually easy to analyze. Reasonable analytical results also can be gained for Phase–Type distributions. Other time behavior that can be found in the literature for TPN's, e. g., Deterministic timing (e. g., [MC87]), mixture of Deterministic and Exponential timing, and interval timing, should result in reasonable results for TDFD's, too.

Many problems that occur during the analysis of TDFD's have been known for a long time when analyzing TPN's. Some of these problems concern the state explosion and undecidability. However, there exists a large number of automated tools that help evaluate, analyze, and solve TPN's. To mention only a few, in [Chi85] a software package is introduced that allows the steady state and transient analysis of generalized Stochastic Petri Nets. [Cum85] describes a software package for the analysis of Stochastic Petri Nets models where transition firing times are distributed as Phase–Type. [Men85] provides a tool for the analysis of TPN's where firing only occurs within the limits of time defined by the interval $[a, b], b \geq a$. In [GM95], TimeNET, a tool especially designed for non–Markovian Stochastic Petri Nets is presented and a comparison with other Petri Net tools is given. It should be possible to reuse and extend methods, algorithms, and tools known from TPN's for TDFD's. Once adapted, one might hopefully automatically evaluate, analyze, and solve these TDFD's. Depending on the types of distributions that are allowed for the firing times, one might consider to provide software that is capable of doing an analytical analysis if a Markov chain or a (semi) Markov process is associated with the TDFD. Or, one might do simulations if everything else fails.

In addition to a quantitative analysis (performance, throughput, average load of a bubble, etc.), mostly affected by the choice of the probability distributions, TDFD's also invite a qualitative analysis (deadlock, reachability, termination, finiteness, liveness, etc.). Some answers to qualitative questions may be gained from the related FDFD (e. g., there is no deadlock state for the TDFD if the FDFD has no deadlock state), but others are not immediately achievable (it is not guaranteed that the TDFD can reach a particular state even though it can be reached for the FDFD). Future research should be directed towards the question how decidability problems for TDFD's and FDFD's are related, similar to the discussion in [God82] where liveness properties of Petri Nets and Timed Petri Nets are compared.

Acknowledgements

Symanzik's research was partially supported by a German "DAAD–Doktorandenstipendium aus Mitteln des zweiten Hochschulsonderprogramms". The authors wish to thank Herbert T. David and Kenneth J. Koehler for many valuable suggestions on the presentation of this topic.

6 STOCHASTIC ANALYSIS OF PERIODIC TIMED DATA FLOW DIAGRAMS WITH MARKOVIAN TRANSITION TIMES

A paper submitted to the IEEE Transactions on Software Engineering

Jürgen Symanzik

Abstract

Timed (or Stochastic) Data Flow Diagrams (TDFD's or SDFD's) introduced in [SB96d] are an extension of the Formalized Data Flow Diagrams, defined in [LWBL96]. This extension allows us to assess the quantitative behavior (e. g., performance, throughput, average load of a bubble, etc.) as well as the qualitative behavior (e. g., deadlock, reachability, termination, finiteness, liveness, etc.), eventually depending on different types of transition times, for the system modeled through the TDFD. In this paper, we consider Markovian transition times for the consumption of in-flow items and for the production of items on the out-flow. Moreover, we require the TDFD to be periodic and irreducible and it must have a finite reachability set. For these models, we have been able to apply an aggregation principle of [Sch84], extended for periodic Markov chains by [Woo93], to efficiently determine stationary probabilities, expected waiting times, and limiting process probabilities.

Keywords

Statistical Software Engineering, Formalized Data Flow Diagrams, Embedded Markov Process, Embedded Markov Chain, Aggregation Principle, Periodicity.

6.1 Introduction

In [SB96d] we introduced Timed (or Stochastic) Data Flow Diagrams (TDFD's or SDFD's) as an extension of Formalized Data Flow Diagrams (FDFD's), defined in [LWBL96]. In SDFD's, time is modeled through the definition of a stochastic time behavior for the consumption of in–flow items as well as a stochastic time behavior for the production of items on the out–flow. We followed the general approach of Stochastic Petri Nets given in [MBB+85] when defining SDFD's.

In this paper we consider one particular subclass of TDFD's, i. e., those that are periodic, irreducible, and have Markovian transition times. We call these SDFD's periodic Markovian Timed Data Flow Diagrams (periodic M–TDFD's). We will demonstrate how stationary probabilities, expected waiting times, and limiting process probabilities can be derived for M–TDFD's. The periodicity of the Markov chain, embedded in the Markov process (which is embedded in the given periodic M–TDFD), plays an important role for a computationally efficient analysis of interesting questions. This analysis is based on the aggregation principle of [Sch84]. Similarly, [Woo93] (Chapter 2) used this aggregation principle and the periodicity of N–stage stochastic service systems (it appears, as pointed out in [Woo93], that [Pat64] first noted this periodicity) to efficiently derive stationary probabilities and limiting process probabilities for 3– and 4–stage Markovian production lines.

Some work has been done to exploit the periodic functioning of Timed Petri Nets. In [Hil90] for example, results have been obtained for the performance evaluation of multi–stage production systems where this periodic functioning often occurs. [Yua86] defines process periods for Petri Nets and uses those to describe the system behavior. However, to our best knowledge, there exists no prior approach to aggregate the state space of periodic Timed Petri Nets or similar computational models in a manner suggested by the aggregation principle of [Sch84] and the extension for periodic Markov chains by [Woo93].

The typical two–step firing behavior of FDFD's has been applied to Timed Petri Nets as well. [RP84] allocates both an enabling time and a firing time to each transition. After a transition is enabled, it has to wait for a time (called the "enabling time") before it absorbs all tokens from its input bag. The tokens remain absorbed for the "firing time" after which the transition places tokens in the appropriate output bag. The idea of alternating enabling and firing time points is also built into the work of [HT91].

The work presented in this paper can not only be applied to periodic M–TDFD's, but it can be directly used for the previously described Petri Nets with alternating enabling and firing time points. However, since our background is in Software Engineering, in particular in "Structured Analysis" (SA) (e. g., [DeM78], [WM85a]) where traditional Data Flow Diagrams (DFD's) are probably the most widely

used specification technique in industry today ([BB93]), we wanted to present our results in this context. Even though little work has been done on TDFD's to date, we want to encourage the reader to consider TDFD's as a helpful model for complex time dependend systems. It does not matter for the stochastic analysis whether the Markov process and Markov chain under consideration result from a TDFD or a Timed Petri Net. Thus, many known methods and results for the stochastic analysis of Timed Petri Nets can be directly applied to TDFD's. Moreover, several advantages of TDFD's over Timed Petri Nets make them the more natural selection for software engineers as it has been pointed out in [SB96d].

The work presented in this paper relates to the new interdisciplinary field "Statistical Software Engineering", introduced in [Nat96]. We use statistical techniques that allow us to reduce computations when we analyze in the Specifications phase of the spiral software development process model ([Nat96], p. 63) whether quantitative requirements of the software system are fulfilled.

In Section 6.2, we will summarize basic definitions required within this paper. Section 6.3 deals with the characterization of periodic FDFD's. In Section 6.4, we demonstrate how to apply the aggregation principle of [Sch84] to periodic and irreducible M–TDFD's with finite reachability set. We conclude this paper with an overview of future work in Section 6.5.

6.2 Definitions

6.2.1 Stochastic Data Flow Diagrams

Data Flow Diagrams have been formalized at multiple places within the technical literature, e. g., in [DeM78], [WM85a], [WM85b], [Har87], [TP89], [You89], [Har92], and [Har96]. Within this paper, we make use of the definitions of Formalized Data Flow Diagrams (FDFD's) developed by Coleman, Wahls, Baker, and Leavens in [Col91], [CB94], [WBL93], and [LWBL96]. In particular for our examples, we use the notation from [LWBL96]. This cited paper also contains a more detailed explanation of the underlying operational semantics of FDFD's and an extended example.

In addition, we need the following defintions from [SB96d]:

Definition (6.2.1.1): A *firing sequence* (*computation sequence*) of an FDFD is a possibly infinite sequence $(b_i, a_i, j_i) \in B \times \{C, P\} \times \mathbb{N}, i \geq 0$, such that, if transition (b_i, a_i, j_i) is fired in state (bm, r, fs), then

$$(fs', r') = \begin{cases} (Consume(b_i))_{j_i}(fs, r), & \text{if } a_i = C \\ (Produce(b_i))_{j_i}(fs, r), & \text{if } a_i = P \end{cases}$$

$$bm'(b_i) = \begin{cases} working, & \text{if } a_i = C \\ idle, & \text{if } a_i = P \end{cases}$$

$$bm'(b) = bm(b) \quad \forall b \in B - \{b_i\}$$

and

$$(bm, r, fs) \rightarrow (bm', r', fs').$$

We introduce the notation $(bm, r, fs)[(b, a, j)]$ to indicate that transition (b, a, j) is fireable in state (bm, r, fs) and $(bm, r, fs)[(b, a, j)](bm', r', fs')$ to indicate that state (bm', r', fs') is reached upon the firing of transition (b, a, j) in state (bm, r, fs).

By induction, we extend this notation for firing sequences:

$$(bm_0, r_0, fs_0)[(b_1, a_1, j_1), \ldots, (b_{n-1}, a_{n-1}, j_{n-1}), (b_n, a_n, j_n)]$$

is used to indicate that transition (b_n, a_n, j_n) is fireable in state $(bm_{n-1}, r_{n-1}, fs_{n-1})$, given that

$$(bm_0, r_0, fs_0)[(b_1, a_1, j_1), \ldots, (b_{n-1}, a_{n-1}, j_{n-1})](bm_{n-1}, r_{n-1}, fs_{n-1})$$

holds. By analogy, we use

$$(bm_0, r_0, fs_0)[(b_1, a_1, j_1), \ldots, (b_n, a_n, j_n)](bm_n, r_n, fs_n)$$

to indicate that state (bm_n, r_n, fs_n) is reached upon the firing of the sequence $(b_1, a_1, j_1), \ldots, (b_n, a_n, j_n)$.

■

Definition (6.2.1.2): The *set of firing sequences* (*set of computation sequences, language*) of an FDFD, denoted by $FS(FDFD, \gamma_{initial})$. is the set containing all firing sequences that are possible for this FDFD, given $\gamma_{initial} = (bm_{initial}, r_{initial}, fs_{initial})$, i. e.,

$$FS(FDFD, \gamma_{initial}) = \{s \mid s \in (B \times \{C, P\} \times \mathbb{N})^* \wedge \gamma_{initial}[s]\}.$$

By analogy, we define

$$FS_i(FDFD, \gamma_{initial}) = \{s \mid s \in (B \times \{C, P\} \times \mathbb{N})^i \wedge \gamma_{initial}[s]\}, i \geq 0,$$

the set of firing sequences of length i when starting in $\gamma_{initial}$. ■

Definition (6.2.1.3): The *Reachability Set* of an FDFD, denoted by $RS(FDFD, \gamma_{initial})$, is the set of states $\gamma = (bm, r, fs)$ that are reachable from $\gamma_{initial} = (bm_{initial}, r_{initial}, fs_{initial})$, i. e.,

$$RS(FDFD, \gamma_{initial}) = \{\gamma \mid \gamma \in \Gamma \wedge \exists s \in FS(FDFD, \gamma_{initial}) : \gamma_{initial}[s]\gamma\}.$$

By analogy, we define

$$RS_i(FDFD, \gamma_{initial}) = \{\gamma \mid \gamma \in \Gamma \wedge \exists s \in FS_i(FDFD, \gamma_{initial}) : \gamma_{initial}[s]\gamma\}, i \geq 0,$$

the set of states that are reachable in i steps when starting in $\gamma_{initial}$. ■

Definition (6.2.1.4): Let $EN(\gamma) \subseteq B \times \{C, P\} \times I\!N$ be the set of transitions that are enabled in state $\gamma = (bm, r, fs)$, i. e.,

$$EN(\gamma) = \{s \mid s \in (B \times \{C, P\} \times I\!N) \wedge \gamma[s]\} = FS_1(FDFD, \gamma).$$ ■

Now, we specialize our definitions from [SB96d] with respect to Markovian transition times.

Definition (6.2.1.5): A *timed firing sequence* (TFS) of an FDFD with initial state $\gamma_{initial}$ is a pair $tfs = (s, \tau)$, where $s \in FS(FDFD, \gamma_{initial})$ and τ is a non–decreasing sequence (of the same length) of real non–negative values representing the instants of firing (called *epochs*) of each transition, such that consecutive transitions (b_i, a_i, j_i) and $(b_{i+1}, a_{i+1}, j_{i+1})$ correspond to ordered epochs $\tau_i \leq \tau_{i+1}$. The time intervals $[\tau_i, \tau_{i+1})$ between consecutive epochs represent the periods in which the FDFD remains in state γ_i (assuming $\tau_0 = 0$). A *history* of the FDFD up to the kth epoch τ_k is denoted by $Z(k)$. ■

Definition (6.2.1.6): A *Markovian Timed Data Flow Diagram* (M–TDFD) is a SDFD with associated FDFD and initial state $\gamma_{initial}$ where the selection of the transition that fires is based on the Race Policy with marginal distributions (that do not depend on state $\underline{\gamma}$ and past history $\underline{Z}$)

$$\phi_i(x) = \phi_i(x \mid \underline{\gamma}, \underline{Z}) = Exp(x; \lambda_i), i = 1, \ldots, \mid EN(\underline{\gamma}) \mid \ \forall \underline{\gamma}\, \forall \underline{Z}$$

and with an initial probability distribution on the Reachability Set $RS(FDFD, \gamma_{initial})$. $Exp(x; \lambda_i)$ represents the Exponential distribution with probability density function $f(x) = \lambda_i \exp(-\lambda_i x) I_{(0,\infty)}(x)$ and cumulative distribution function $F(x) = (1 - \exp(-\lambda_i x)) I_{(0,\infty)}(x)$ for $\lambda_i > 0$. ■

Because of the memoryless property of the Exponential distribution, we do not have to distinguish among the possible cases introduced in [SB96d] how to deal with the past history $\underline{Z}$. We have the same behavior for Resampling, Work Age Memory, and Enabling Age Memory.

This definition introduces the *embedded Markov process* of the M–TDFD with a one–on–one mapping between the discrete state space and the reachability set $RS(FDFD, \gamma_{initial})$. We refer to this state

space together with the rules for state changes (implied by the mappings *Enabled*, *Consume*, and *Produce* of the M–TDFD) as the *embedded Markov chain* of the M–TDFD. For the initial probability distribution on $RS(FDFD, \gamma_{initial})$, we assume that $Pr(\text{system is in state } \gamma_{initial} \text{ at time } \tau_0 = 0) = 1$.

6.2.2 Periodic Markov Chains

In this section we will summarize definitions and theorems on periodic Markov chains given in [IM76], Chapters 2 and 3. It is assumed that the reader is familiar with the basic notations for Markov chains.

Definition (6.2.2.1): A subset, C, of the state space, S is called *closed* if $p_{ik} = 0$ for all $i \in C$ and $k \notin C$. If a closed set consists of a single state, then that state is called an *absorbing state.* ■

Definition (6.2.2.2): A Markov chain is called *irreducible* if there exists no nonempty closed set other than S itself. If S has a proper closed subset, it is called *reducible.* ■

Definition (6.2.2.3): Two states, i and j, are said to *intercommunicate* if for some $n \geq 0$, $p_{ij}^{(n)} > 0$ and for some $m \geq 0$, $p_{ji}^{(m)} > 0$. ■

Theorem (6.2.2.4): A Markov chain is irreducible if and only if all pairs of states intercommunicate.

Definition (6.2.2.5): State j has *period* d if the following two conditions hold:

(i) $p_{jj}^{(n)} = 0$ unless $n = md$ for some positive integer m and

(ii) d is the largest integer with property (i).

State j is called *aperiodic* when $d = 1$. ■

Theorem (6.2.2.6): State j has period d if and only if d is the greatest common divisor of all those n's for which $p_{jj}^{(n)} > 0$ (that is, $d = G.C.D.\{n \mid p_{jj}^{(n)} > 0\}$).

Lemma (6.2.2.7): The state space of a periodic irreducible Markov chain of period d can be partitioned into d disjoint classes $D_0, D_1, \ldots, D_{d-1}$ such that from D_j the chain goes, in the next step,

to D_{j+1} for $j = 0, 1, \ldots, d-2$. From D_{d-1} the chain returns in the next step to D_0. ■

Finally, we indicate the following Theorem, introduced as Proposition 6–28 and proved in [KSK76].

Theorem (6.2.2.8): The period of a recurrent chain for the state i is a constant independent of the state i.

6.2.3 Periodic Formalized Data Flow Diagrams

Similiar to Subsection 6.2.2, we now define related terms for an FDFD with initial state $\gamma_{initial}$. It should be obvious that there exists a one–on–one mapping between the reachability set $RS(FDFD, \gamma_{initial})$ of an FDFD and the discrete state space of a Markov chain. We refer to this state space together with the rules for state changes (implied by the mappings *Enabled*, *Consume*, and *Produce* of the FDFD) as the *embedded Markov chain* of the FDFD.

Definition (6.2.3.1): A subset C, of the reachability set, $RS(FDFD, \gamma_{initial})$, is called *closed* if for all $\gamma_i \in C$ and $\gamma_k \notin C$ there exists no transition $s \in (B \times \{C, P\} \times \mathbb{N})$ such that $\gamma_i[s]\gamma_k$. If a closed set consists of a single state, then that state is called a *deadlock* state. ■

Definition (6.2.3.2): An FDFD with initial state $\gamma_{initial}$ is called *irreducible* if there exists no nonempty closed set other than $RS(FDFD, \gamma_{initial})$ itself. If $RS(FDFD, \gamma_{initial})$ has a proper closed subset, it is called *reducible*. ■

Definition (6.2.3.3): Two states, γ_i and $\gamma_j \in RS(FDFD, \gamma_{initial})$, are said to *intercommunicate* if for some firing sequences s and t, $\gamma_i[s]\gamma_j$ and $\gamma_j[t]\gamma_i$. ■

Theorem (6.2.3.4): An FDFD with initial state $\gamma_{initial}$ is irreducible if and only if all pairs of states intercommunicate.

Proof: Follows directly from the embedded Markov chain. ■

Definition (6.2.3.5): A state $\gamma_i \in RS(FDFD, \gamma_{initial})$ has *period* d if the following two conditions hold:

(i) $\gamma_i[s]\gamma_i$ does not hold unless $s \in FS_n(FDFD, \gamma_i)$ where $n = md$ for some positive integer m and

(ii) d is the largest integer with property (i). ■

Note that the period d of a state γ_i is related to the times at which the FDFD might return to state γ_i. It does not mean that the FDFD with current state γ_i can return to this state upon the firing of exactly d transitions nor does it mean that the $FDFD$ will ever return to this state. Also, we do not have to define an aperiodic state γ_i, where $d = 1$. This case can never happen as we show in the next theorem.

Definition (6.2.3.6): An FDFD with initial state $\gamma_{initial}$ is *periodic with period d* if all of its states $\gamma \in RS(FDFD, \gamma_{initial})$ have period d. ■

Theorem (6.2.3.7): An FDFD with initial state $\gamma_{initial}$ is either aperiodic or it is periodic with period $d \geq 2$, where d is even.

Proof: It is impossible that the state does not change upon the firing of a transition. Even if nothing is consumed and nothing is produced upon the firing of a transition, at least one bubble changes its *BubbleMode*. Every bubble has to move from *idle* to *working* (*working* to *idle*) before it can return to *idle* (*working*). Therefore, $d \geq 2$. Through the execution of every transition, the *BubbleMode* of exactly one bubble is altered. Thus, after an odd number of transitions, at least one bubble has a *BubbleMode* different from its *BubbleMode* in the starting state. Thus, d is even. However, if the FDFD contains a deadlock state, if it can never return to some previously reached state, or if some states have different periods, then it is aperiodic. ■

Corollary (6.2.3.8): An irreducible FDFD with initial state $\gamma_{initial}$ and finite reachability set is periodic with period $d \geq 2$, where d is even.

Proof: Since the FDFD is irreducible, it has no deadlock state and all pairs of states intercommunicate. It is possible to return to any state in $RS(FDFD, \gamma_{initial})$. This means, the embedded Markov chain is recurrent since the state space (the reachability set of the FDFD) also is finite. According to Theorem (6.2.2.8), all states have the same period d. But then, as we have seen in the previous Theorem, $d \geq 2$ and d is even. ■

6.3 Characterization of Periodic FDFD's

For the results in Section 6.4 we will assume that a given M-TDFD with initial state $\gamma_{initial}$ is periodic, irreducible, and has a finite reachability set. However, what does a M-TDFD with these

features look like, what are the necessary criteria it has to fulfill? To answer these questions, we will first demonstrate the unpredictable behavior of FDFD's in Subsection 6.3.1 and indicate how the period d can be determined for FDFD's with finite reachability set in Subsection 6.3.2.

6.3.1 Unpredictable Behavior of FDFD's

As we have seen in [SB96a], FDFD's are computationally equivalent to Turing Machines. This implies that all interesting decidability problems such as reachability, termination, deadlock and liveness properties, and finiteness, that are undecidable for Turing Machines are also undecidable for FDFD's. Unfortunately, the questions whether an FDFD is periodic, irreducible, and has a finite reachability set are directly related to these decidability problems.

In the next example, we will see that the question whether an FDFD is periodic, irreducible, and has a finite reachability set does not only depend on the structure and the mappings *Enabled*, *Consume*, and *Produce*, but it depends on the initial state $\gamma_{initial}$ as well. Moreover, an FDFD with initial state $\gamma_{initial_1}$ might be periodic, irreducible, and might have a finite reachability set, but the same FDFD with initial state $\gamma_{initial_2}$ may not have any of these features.

Example (6.3.1.1): This example shows an FDFD whose features highly depend on its initial state $\gamma_{initial}$.

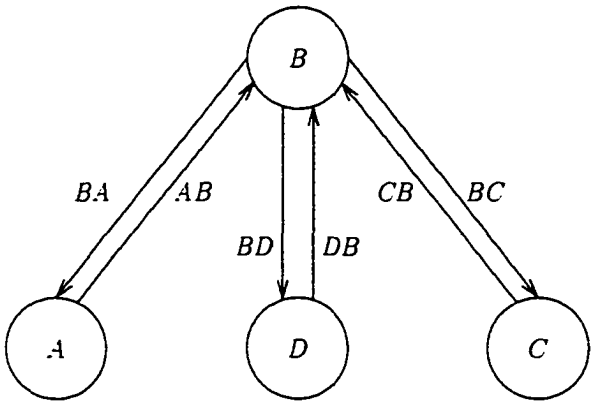

Figure 6.1: Example of an FDFD Highly Depending on $\gamma_{initial}$.

The mappings *Enabled*, *Consume*, and *Produce* for the FDFD shown in Figure 6.1 are defined as:

$$Enabled(A) = \lambda fs\ .\ (\neg IsEmpty(BA) \wedge Head(fs(BA)) = d)$$

$$Enabled(B) = \lambda fs\ .\ (\neg IsEmpty(AB) \wedge Head(fs(AB)) = a)$$
$$\vee\ (\neg IsEmpty(CB) \wedge Head(fs(CB)) = c)$$

$$\vee\ (\neg IsEmpty(AB) \wedge Head(fs(AB)) = a$$
$$\wedge \neg IsEmpty(CB) \wedge Head(fs(CB)) = c)$$
$$\vee\ (\neg IsEmpty(AB) \wedge Head(fs(AB)) = a$$
$$\wedge \neg IsEmpty(DB) \wedge Head(fs(DB)) = f)$$

$Enabled(C) = \lambda fs\ .\ (\neg IsEmpty(BC) \wedge Head(fs(BC)) = b)$

$Enabled(D) = \lambda fs\ .\ (\neg IsEmpty(BD) \wedge Head(fs(BD)) = e)$

Transition

$Consume(A) = \lambda(fs, r)\ .$

{**if** $(\neg IsEmpty(BA) \wedge Head(fs(BA)) = d)$
then $In(BA, A)(fs, r)$ (A, C, 1)
fi
}

$Consume(B) = \lambda(fs, r)\ .$

{**if** $(\neg IsEmpty(AB) \wedge Head(fs(AB)) = a)$
then $In(AB, B)(fs, r)$ (B, C, 1)
fi,
if $(\neg IsEmpty(CB) \wedge Head(fs(CB)) = c)$
then $In(CB, B)(fs, r)$ (B, C, 2)
fi,
if $(\neg IsEmpty(AB) \wedge Head(fs(AB)) = a$
$\wedge \neg IsEmpty(CB) \wedge Head(fs(CB)) = c)$
then $In(AB, B)(In(CB, B)(fs, r))$ (B, C, 3)
fi,
if $(\neg IsEmpty(AB) \wedge Head(fs(AB)) = a$
$\wedge \neg IsEmpty(DB) \wedge Head(fs(DB)) = f)$
then $In(AB, B)(In(DB, B)(fs, r))$ (B, C, 4)
fi
}

$Consume(C) = \lambda(fs, r)\ .$

{**if** $(\neg IsEmpty(BC) \wedge Head(fs(BC)) = b)$
then $In(BC, C)(fs, r)$ (C, C, 1)
fi

}

$Consume(D) = \lambda(fs, r)$.

{**if** $(\neg IsEmpty(BD) \wedge Head(fs(BD)) = e)$

then $In(BD, D)(fs, r)$ (D, C, 1)

fi

}

$Produce(A) = \lambda(fs, r)$.

{**if** $r(A)(BA) = d$

then $Out(a, AB, A)(fs, r)$ (A, P, 1)

fi

}

$Produce(B) = \lambda(fs, r)$.

{**if** $r(B)(AB) = a$

then $Out(b, BC, B)(fs, r)$ (B, P, 1)

fi,

if $r(B)(CB) = c$

then $Out(d, BA, B)(fs, r)$ (B, P, 2)

fi,

if $r(B)(AB) = a \wedge r(B)(CB) = c$

then $Out(e, BD, B)(fs, r)$ (B, P, 3)

fi,

if $r(B)(AB) = a \wedge r(B)(DB) = f$

then $Out(b, BC, B)(fs, r)$ (B, P, 4)

fi

}

$Produce(C) = \lambda(fs, r)$.

{**if** $r(C)(BC) = b$

then $Out(c, CB, C)(fs, r)$ (C, P, 1)

fi

}

$Produce(D) = \lambda(fs, r)$.

{**if** $r(D)(BD) = e$

then $Out(f, DB, D)(fs, r)$ (D, P, 1)

fi

}

We consider three initial states

$$\gamma_{initial_1} = ((idle, idle, idle, idle), (\perp, \perp, \perp, \perp, \perp, \perp), ((a), (), (), (), (), ())),$$

$$\gamma_{initial_2} = ((idle, idle, idle, idle), (\perp, \perp, \perp, \perp, \perp, \perp), ((aa), (), (), (), (), ())),$$

and

$$\gamma_{initial_3} = ((idle, idle, idle, idle), (\perp, \perp, \perp, \perp, \perp, \perp), ((aaa), (), (), (), (), ())).$$

Note that the only difference between these initial states is the number of times the *OBJECT* "a" initially appears on flow AB. This notation indicates

$$(bm_{initial}, r_{inital}, fs_{inital}) =$$
$$((bm(A), bm(B), bm(C), bm(D)),$$
$$(r(B)(AB), r(C)(BC), r(B)(CB), r(A)(BA), r(D)(BD), r(B)(DB)),$$
$$(fs(AB), fs(BC), fs(CB), fs(BA), fs(BD), fs(DB))).$$

For $\gamma_{initial_1}$, the reachability set $RS(FDFD, \gamma_{initial_1})$ consists only of the following eight states $\gamma_1, \ldots, \gamma_8$:

- $\gamma_1 = ((idle, idle, idle, idle), (\perp, \perp, \perp, \perp, \perp, \perp), ((a), (), (), (), (), ())) = \gamma_{initial_1}$
- $\gamma_2 = ((idle, working, idle, idle), (a, \perp, \perp, \perp, \perp, \perp), ((), (), (), (), (), ()))$
- $\gamma_3 = ((idle, idle, idle, idle), (\perp, \perp, \perp, \perp, \perp, \perp), ((), (b), (), (), (), ()))$
- $\gamma_4 = ((idle, idle, working, idle), (\perp, b, \perp, \perp, \perp, \perp), ((), (), (), (), (), ()))$
- $\gamma_5 = ((idle, idle, idle, idle), (\perp, \perp, \perp, \perp, \perp, \perp), ((), (), (c), (), (), ()))$
- $\gamma_6 = ((idle, working, idle, idle), (\perp, \perp, c, \perp, \perp, \perp), ((), (), (), (), (), ()))$
- $\gamma_7 = ((idle, idle, idle, idle), (\perp, \perp, \perp, \perp, \perp, \perp), ((), (), (), (d), (), ()))$
- $\gamma_8 = ((idle, idle, working, idle), (\perp, \perp, \perp, d, \perp, \perp), ((), (), (), (), (), ()))$

At any time during the execution of the FDFD, we have $\mid EN(\gamma_i) \mid = 1, i = 1, \ldots, 8$, i. e., the FDFD behaves deterministically. Therefore, upon the execution of the firing sequence

$$s_1 = ((B,C,1),(B,P,1),(C,C,1),(C,P,1),(B,C,2),(B,P,2),(A,C,1),(A,P,1))$$

the FDFD returns to $\gamma_{initial_1}$ when started in $\gamma_{initial_1}$, i. e., $\gamma_{initial_1}[s_1]\gamma_{initial_1}$. Trivially, the FDFD with $\gamma_{initial_1}$ is irreducible, periodic with period 8, and has a finite reachability set of size 8.

Now we consider the FDFD with $\gamma_{initial_2}$. Since only an *OBJECT* on a flow has been replicated but no *OBJECT* has been removed in comparision with $\gamma_{initial_1}$, it should be obvious that $\gamma_{initial_2}[s_1]\gamma_{initial_2}$ holds. But, is the FDFD with $\gamma_{initial_2}$ still irreducible and periodic? Consider

$$s_2 = ((B,C,1),(B,P,1),(C,C,1),(C,P,1),(B,C,3),(B,P,3),(D,C,1),(D,P,1)).$$

We have $\gamma_{initial_2}[s_2]\gamma_{dead}$, where $\gamma_{dead} = ((idle, idle, idle, idle),(\perp,\perp,\perp,\perp,\perp,\perp),((),(),(),(),(),(f)))$ denotes a deadlock state. Hence, the FDFD with $\gamma_{initial_2}$ is not irreducible. γ_{dead} has no period d and thus the FDFD with $\gamma_{initial_2}$ is not periodic.

Finally, we consider the FDFD with $\gamma_{initial_3}$. Again, one *OBJECT* on a flow has been replicated twice and no *OBJECT* has been removed in comparision with $\gamma_{initial_1}$. We consider s_2 first. We have $\gamma_{initial_3}[s_2]\gamma_{not\_dead}$, where $\gamma_{not\_dead} = ((idle, idle, idle, idle),(\perp,\perp,\perp,\perp,\perp,\perp),((a),(),(),(),(),(f)))$ represents a state that is not a deadlock state. Instead, we have $\gamma_{not\_dead}[(B,C,4),(B,P,4)]\gamma_3$, where γ_3 is identical to the state of the same name in $RS(FDFD, \gamma_{initial_1})$. Once γ_3 has been reached, only those states can be reached that are also reachable for the FDFD with $\gamma_{initial_1}$. But none of these states intercommunicates with $\gamma_{initial_3}$ for example, thus the FDFD with $\gamma_{initial_3}$ can not be irreducible. We leave it to the reader to determine whether the FDFD with $\gamma_{initial_3}$ is periodic and which deadlock states can possibly be reached. This exmple illustrates how such small changes in the initial state result in quite different behavior. We leave it to the reader to imagine how inpredictably a more complex FDFD might behave. ■

Based on the previous example, it becomes obvious that our initial assumption that the period d (if the FDFD is periodic at all) might depend on the number of bubbles and flows does not hold at all. We found several examples where the period d is proportional to $(2 \cdot \#bubbles)$, allowing each of the bubbles to alter between *idle* and *working* for the same number of times. However, we can not generalize from these examples to the general behavior of FDFD's. Especially since the period does not only depend on the structure but on the initial state as well. This is reasonable when recalling the fact that FDFD's have the same computational power as Turing Machines. Anything can happen.

6.3.2 Determination of d

As we already pointed out earlier, FDFD's are computationally equivalent to Turing Machines ([SB96a]). Therefore, we can not generally answer the questions whether an FDFD is periodic, irreducible, and has a finite reachability set. However, we can indicate an algorithm that determines whether the reachability set is finite or terminates after a fixed number of steps if the reachability set has not been fully exploited by then. If the reachability set is finite, we can determine whether the FDFD is irreducible, and if so, what period d it has.

First we want to introduce a definition from graph theory (e. g., [AHU74], p. 189):

Definition (6.3.2.1): Let $G = (V, E)$ be a directed graph. We can partition V into equivalence classes, $V_i, 1 \leq i \leq r$, such that vertices v and w are equivalent if and only if there is a path from v to w and a path from w to v. Let $E_i, 1 \leq i \leq r$, be the set of edges connecting the pairs of vertices in V_i. The graphs $G_i = (V_i, E_i)$ are called the *strongly connected components* of G. Even though every vertex of G is in some V_i, G may have edges not in any E_i. A graph is said to be *strongly connected* if it has only one strongly connected component. ■

Obviously, the question whether an FDFD with initial state $\gamma_{initial}$ is irreducible is equivalent to the question whether its reachability graph is strongly connected.

We can summarize the procedure that eventually returns the period d in four steps:

Step 1 : Determine the reachability set $RS(FDFD, \gamma_{initial})$

This can be done using the following breadth-first algorithm:

$i = 0$; $New_i = \{\gamma_{initial}\}$; $Reached_i = New_i$; $Examine_i = New_i$
while ($i \leq$ MAXSTEPS and $Examine_i \neq \{\}$)
{
$New_{i+1} = \bigcup_{\gamma \in Examine_i} RS_1(FDFD, \gamma)$ % all states reachable in one more step
$Reached_{i+1} = Reached_i \cup New_{i+1}$ % all states reached so far
$Examine_{i+1} = New_{i+1} - Reached_i$ % all states not yet examined
$i = i + 1$
}
if $Examine_i = \{\}$ exit "Reachability set is finite."
else exit "No solution found."

We stop if we find no solution.

Step 2 : Determine whether the reachability graph is strongly connected

Let $G = (V, E)$ with $V = RS(FDFD, \gamma_{initial})$ and E the set of edges implied by the mappings *Consume* and *Produce* be the (directed) reachability graph of the related FDFD with initial state $\gamma_{initial}$. Let n = number of vertices = $\mid RS(FDFD, \gamma_{initial}) \mid$ and e = number of edges. We can apply an algorithm that finds the strongly connected components of G. For example, Algorithm 5.4 in [AHU74], p. 193, performs this task in $O(MAX(n, e))$ time. We stop if G is not strongly connected, i. e., if the FDFD with $\gamma_{initial}$ is not irreducible.

Step 3 : Determine the shortest return path

For every $\gamma_i \in RS(FDFD, \gamma_{initial}), i = 1, \ldots, n$, we determine the shortest path from γ_i to γ_i with length $d_i > 0$. This can be done applying Dijkstra's Algorithm (e. g., Algorithm 5.6 in [AHU74], p. 207, or Section 6.4 in [PS82]) to every γ_i in time $O(n^2)$. We could also use the Floyd–Warshall Algorithm (e. g., Section 6.5 in [PS82]) that finds the shortest paths between all pairs of nodes in $O(n^3)$ time.

Step 4 : Determine the period d

According to Corollary (6.2.3.8) the FDFD with $\gamma_{initial}$ is periodic with period $d \geq 2$, where d is even. We can determine d as $G.C.D.\{d_i \mid i = 1, \ldots, n\}$.

Obviously, this 4–step approach is not the most efficient one. For example, we do not really need Dijkstra's Algorithm or the Floyd–Warshall Algorithm to determine the shortest path since the cost for each step is the same, no matter which edge is selected. Moreover, it should be possible to combine Steps 1 to 3 into a more efficient algorithm. However, this goes beyond the scope of this paper.

6.4 Analysis of Periodic M–TDFD's

6.4.1 The Aggregation Principle

The idea presented in the following extract from [Sch84] is commonly referred to as the *aggregation principle*:

> "Let S be a finite or countably infinite set and $X = \{X_n; n \geq 0\}$ a homogeneous irreducible recurrent Markov chain on S with transition matrix $P = (p_{ij})$. Let S' be a nonempty subset of S, and denote by $n_1, n_2, \ldots$ the successive random times at which X is visiting S'. Then

$X' = \{X_{n_1}, X_{n_2}, \ldots\}$ is a homogeneous irreducible recurrent Markov chain on S' ([KSK76], p. 164[1]). Its transition matrix, P', is given by

$$(1.1) \quad p'_{ij} = p_{ij} + \sum_{k \in S - S'} p_{ik} r_{kj}, \quad i, j \in S',$$

where $r_{kj}, k \in S - S', j \in S'$, is the probability that X will first hit S' at state j given start in k. Likewise, p'_{ij} is the probability for X to first reenter S' at j given start in i. If X is ergodic, so is X' (but not vice versa). X' will be said to arise from X by watching X on S', only. The equations (1.2) $x = xP$ and (1.3) $x' = x'P'$, x and x' denoting row vectors, possess strictly positive solutions p, p' which are unique up to multiplicative constants, and for which (1.4) $p'_i = cp_i, i \in S'$, c a constant ([KSK76], p. 164[1]). The p, p' shall be assumed to denote probability vectors in the ergodic case."

6.4.2 Application to Periodic M–TDFD's

As pointed out in [Woo93], in the case of a periodic Markov chain, if S' is taken as a periodic subset of S, then the constant c in [Sch84] (1.4) is simply the period d of the Markov chain.

We can summarize the process to analyze a periodic and irreducible M–TDFD of period d with initial state $\gamma_{initial}$ and finite reachability set $RS(FDFD, \gamma_{initial})$ in the following algorithm:

(i) Determine the d equivalence classes of states of the FDFD associated with M–TDFD (the periodic subsets) $S_1, \ldots, S_d$. It is

$$S_j = \bigcup_{i=0}^{ceil(|RS(FDFD,\gamma_{initial})|/d)-1} RS_{id+j}(FDFD, \gamma_{initial}), j = 1, \ldots, d,$$

and

$$S = \bigcup_{j=1}^{d} S_j = RS(FDFD, \gamma_{initial}).$$

(ii) Determine P, with columns and rows representing states $\gamma_1, \ldots, \gamma_n$, $n = \mid RS(FDFD, \gamma_{initial}) \mid$, ordered according to the periodic subsets, starting with $S_d, S_1, \ldots, S_{d-1}$. It is

$$p_{ij} = \frac{\lambda_k}{\sum_{l=1}^{m} \lambda_l}, i, j = 1, \ldots, n,$$

such that $\gamma_i[(b_k, a_k, j_k)]\gamma_j$ and $\gamma_i[(b_l, a_l, j_l)], l = 1, \ldots, m$, where $m = \mid EN(\gamma_i) \mid$ and $\lambda_k, \lambda_1, \ldots, \lambda_m$ are the rates of the Exponential distributions related to the transitions $(b_k, a_k, j_k), (b_1, a_1, j_1), \ldots, (b_m, a_m, j_m)$, respectively.

[1] The reference [KSK76], p. 164, refers to Exercise 5 on page 164. In addition, the definition of P^E on page 133 and Lemma 6–6 on page 134 of the same reference are required to understand the reasoning in [Sch84].

(iii) Let S' be any of the periodic subsets $S_1, \ldots, S_d$ such that $\mid S' \mid \leq \mid S_j \mid \quad \forall j = 1, \ldots, d$. Let $\Delta S = S - S'$. Derive P' according to [Sch84] (1.1). r_{kj}, related to states $\gamma_k \in \Delta S, \gamma_j \in S'$, can be determined via the products of $p_{ij}s$ related to a firing sequence from state γ_k to state γ_j that does not reach any other state $\delta \in S'$, summed up over all possible firing sequences of this type. Note that all firing sequences to be considered are those with $d-1$ or less steps. Now, solve $x' = x'P'$ where solutions p' are strictly positive and can be normalized such that $p'\mathbf{1} = 1$, where $\mathbf{1}$ represents a vector of all 1's.

(iv) From [Sch84] (1.4) and by using P, we get the stationary probabilities p:

$$\begin{aligned} p_i &= \frac{1}{d} p'_i \quad \forall i : \gamma_i \in S' \\ p_i &= \frac{1}{d} \sum_{j:\gamma_j \in S'} p'_j r_{ji} \quad \forall i : \gamma_i \in \Delta S \end{aligned}$$

(v) The expected waiting times Ψ_i in the states $\gamma_i, i = 1, \ldots, n$, can be computed as

$$\Psi_i = \frac{1}{\sum_{l=1}^{m} \lambda_l}, i = 1, \ldots, n,$$

where $m = \mid EN(\gamma_i) \mid$ and $\lambda_1, \ldots, \lambda_m$ are the rates of the related Exponential distributions (as in (ii) above).

(vi) Based on the specialisation in [Woo93] for the general case of ergodic stationary semi Markov processes with countable state space ([AD88]), we can compute the limiting process probabilities $\Pi_i = \lim_{t \to \infty} P_i(t)$ of the states $\gamma_i, i = 1, \ldots, n$, i. e., the probability that the system is in the state γ_i for $t \to \infty$, as

$$\Pi_i = \frac{p_i \Psi_i}{\sum_{j=1}^{n} p_j \Psi_j}, i = 1, \ldots, n.$$

6.4.3 An Example

This example of an M–TDFD represents a Producer/Consumer problem with bounded buffer of size 2 (Figure 6.2). This means, the Producer can only produce two more items than the Consumer has consumed.

The mappings *Enabled*, *Consume*, and *Produce* are defined as:

$$Enabled(P) = \lambda fs \,.\, (\neg IsEmpty(Done) \wedge Head(fs(Done)) = Yes)$$

$$Enabled(C) = \lambda fs \,.\, (\neg IsEmpty(f) \wedge Head(fs(f)) = 1)$$

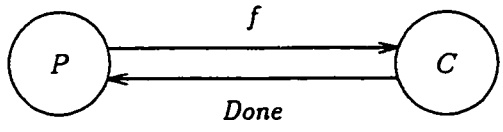

Figure 6.2: Example of a Periodic and Irreducible M–TDFD.

	Transition	Rate
$Consume(P) = \lambda(fs, r)$.		
{**if** $(\neg IsEmpty(Done) \wedge Head(fs(Done)) = Yes)$		
then $In(Done, P)(fs, r)$	(P, C, 1)	λ_1
fi		
}		
$Consume(C) = \lambda(fs, r)$.		
{**if** $(\neg IsEmpty(f) \wedge Head(fs(f)) = 1)$		
then $In(f, C)(fs, r)$	(C, C, 1)	λ_2
fi		
}		
$Produce(P) = \lambda(fs, r)$.		
{**if** $r(C)(Done) = Yes$		
then $Out(1, f, P)(fs, r)$	(P, P, 1)	λ_3
fi		
}		
$Produce(C) = \lambda(fs, r)$.		
{**if** $r(C)(f) = 1$		
then $Out(Yes, Done, C)(fs, r)$	(C, P, 1)	λ_4
fi		
}		

Initially, we have

$$\gamma_1 = \gamma_{initial} = (bm_{initial}, r_{inital}, fs_{inital}) = ((idle, idle), (\perp, \perp), ((), (Yes\, Yes))).$$

This notation indicates

$$((bm(P), bm(C)), (r(P)(Done), r(C)(f)), (fs(f), fs(Done))).$$

The reachability set $RS(FDFD, \gamma_{initial})$ consists of the following eight states:

- γ_1 = ((*idle*, *idle*), ($\perp$, $\perp$), ((), (*Yes Yes*)))
- γ_4 = ((*working*, *idle*), (*Yes*, $\perp$), ((), (*Yes*)))
- γ_6 = ((*idle*, *idle*), ($\perp$, $\perp$), ((1), (*Yes*)))
- γ_7 = ((*working*, *idle*), (Yes, $\perp$), ((1), ()))
- γ_8 = ((*idle*, *working*), ($\perp$, 1), ((), (*Yes*)))
- γ_2 = ((*idle*, *idle*), ($\perp$, $\perp$), ((1 1), ()))
- γ_3 = ((*working*, *working*), (*Yes*, 1), ((), ()))
- γ_5 = ((*idle*, *working*), ($\perp$, 1), ((1), ()))

According to Step 1 in Subsection 6.3.2 the reachability set has been gained in the following way:

i	New	Reached	Examine
0	$\{\gamma_1\}$	$\{\gamma_1\}$	$\{\gamma_1\}$
1	$\{\gamma_4\}$	$\{\gamma_1, \gamma_4\}$	$\{\gamma_4\}$
2	$\{\gamma_6\}$	$\{\gamma_1, \gamma_4, \gamma_6\}$	$\{\gamma_6\}$
3	$\{\gamma_7, \gamma_8\}$	$\{\gamma_1, \gamma_4, \gamma_6, \gamma_7, \gamma_8\}$	$\{\gamma_7, \gamma_8\}$
4	$\{\gamma_2, \gamma_3, \gamma_1\}$	$\{\gamma_1, \gamma_4, \gamma_6, \gamma_7, \gamma_8, \gamma_2, \gamma_3\}$	$\{\gamma_2, \gamma_3\}$
5	$\{\gamma_5, \gamma_4\}$	$\{\gamma_1, \gamma_4, \gamma_6, \gamma_7, \gamma_8, \gamma_2, \gamma_3, \gamma_5\}$	$\{\gamma_5\}$
6	$\{\gamma_6\}$	$\{\gamma_1, \gamma_4, \gamma_6, \gamma_7, \gamma_8, \gamma_2, \gamma_3, \gamma_5\}$	$\{\}$

Following Steps 2 to 4 in Subsection 6.3.2 reveals that the FDFD with $\gamma_{initial}$ is irreducible and periodic with period $d = 4$. The same can be seen in the reachability graph (Figure 6.3).

We follow the algorithm from Subsection 6.4.2 to further analyze this M–TDFD:

(i) The $d = 4$ equivalence classes of states are:

- $S_1 = \{\gamma_4, \gamma_5\}$
- $S_2 = \{\gamma_6\}$
- $S_3 = \{\gamma_7, \gamma_8\}$

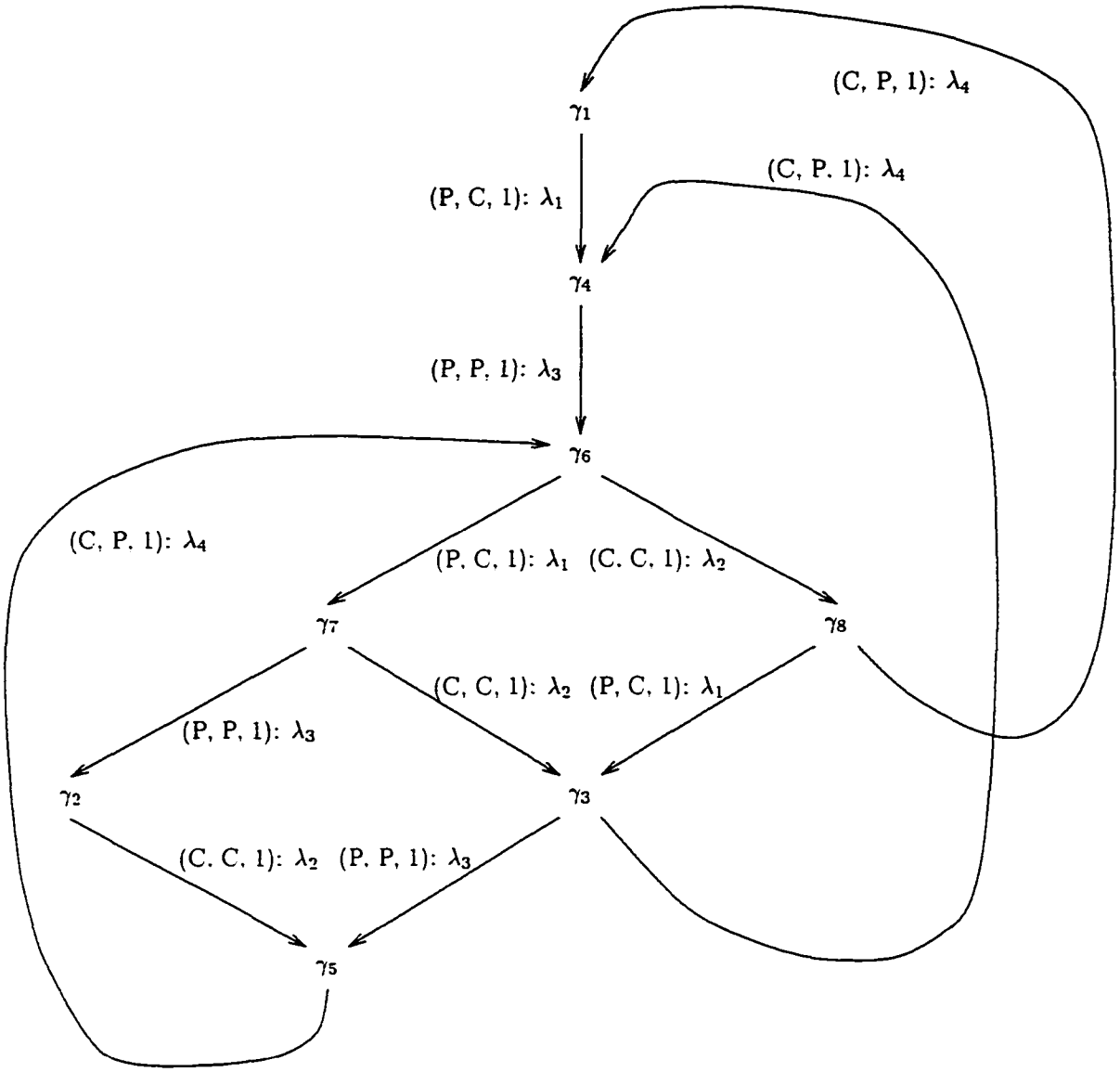

Figure 6.3: Reachability Graph of a Periodic and Irreducible M–TDFD.

- $S_4 = \{\gamma_1, \gamma_2, \gamma_3\}$
- $S = \{\gamma_1, \ldots, \gamma_8\}$

(ii) P can be determined using the reachability graph. It is arranged such that γ_1 appears in the first row/column and γ_8 appears in the last row/column:

$$P = \left(\begin{array}{ccc|cc|c|cc} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{\lambda_4}{\lambda_3+\lambda_4} & \frac{\lambda_3}{\lambda_3+\lambda_4} & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & \frac{\lambda_1}{\lambda_1+\lambda_2} & \frac{\lambda_2}{\lambda_1+\lambda_2} \\ \hline 0 & \frac{\lambda_3}{\lambda_2+\lambda_3} & \frac{\lambda_2}{\lambda_2+\lambda_3} & 0 & 0 & 0 & 0 & 0 \\ \frac{\lambda_4}{\lambda_1+\lambda_4} & 0 & \frac{\lambda_1}{\lambda_1+\lambda_4} & 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

(iii) We select $S' = S_2 = \{\gamma_6\}$. Then $\Delta S = \{\gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5, \gamma_7, \gamma_8\}$. We have

$$\begin{aligned} P' &= .(p'_{66}) \\ &= \left(p_{66} + \sum_{k \in \Delta S} p_{6k} r_{k6}\right) \\ &= (0 + p_{67} r_{76} + p_{68} r_{86}) \\ &= \left(\frac{\lambda_1}{\lambda_1+\lambda_2}\left(\frac{\lambda_3}{\lambda_2+\lambda_3} \cdot 1 \cdot 1 + \frac{\lambda_2}{\lambda_2+\lambda_3} \cdot \frac{\lambda_4}{\lambda_3+\lambda_4} \cdot 1 + \frac{\lambda_2}{\lambda_2+\lambda_3} \cdot \frac{\lambda_3}{\lambda_3+\lambda_4} \cdot 1\right)\right. \\ &\quad \left. + \frac{\lambda_2}{\lambda_1+\lambda_2}\left(\frac{\lambda_4}{\lambda_1+\lambda_4} \cdot 1 \cdot 1 + \frac{\lambda_1}{\lambda_1+\lambda_4} \cdot \frac{\lambda_4}{\lambda_3+\lambda_4} \cdot 1 + \frac{\lambda_1}{\lambda_1+\lambda_4} \cdot \frac{\lambda_3}{\lambda_3+\lambda_4} \cdot 1\right)\right) \\ &= (1) \end{aligned}$$

and $p' = (1)$.

(iv) The stationary probabilities p are:

$$\begin{aligned} p_6 &= \frac{1}{4} \\ p_1 &= \frac{1}{4} \frac{\lambda_2}{\lambda_1+\lambda_2} \frac{\lambda_4}{\lambda_1+\lambda_4} \\ p_2 &= \frac{1}{4} \frac{\lambda_1}{\lambda_1+\lambda_2} \frac{\lambda_3}{\lambda_2+\lambda_3} \\ p_3 &= \frac{1}{4}\left(\frac{\lambda_1}{\lambda_1+\lambda_2} \frac{\lambda_2}{\lambda_2+\lambda_3} + \frac{\lambda_2}{\lambda_1+\lambda_2} \frac{\lambda_1}{\lambda_1+\lambda_4}\right) \\ p_4 &= \frac{1}{4}\left(\frac{\lambda_1}{\lambda_1+\lambda_2} \frac{\lambda_2}{\lambda_2+\lambda_3} \frac{\lambda_4}{\lambda_3+\lambda_4} + \frac{\lambda_2}{\lambda_1+\lambda_2} \frac{\lambda_1}{\lambda_1+\lambda_4} \frac{\lambda_4}{\lambda_3+\lambda_4} + \frac{\lambda_2}{\lambda_1+\lambda_2} \frac{\lambda_4}{\lambda_1+\lambda_4} \cdot 1\right) \\ p_5 &= \frac{1}{4}\left(\frac{\lambda_1}{\lambda_1+\lambda_2} \frac{\lambda_3}{\lambda_2+\lambda_3} \cdot 1 + \frac{\lambda_1}{\lambda_1+\lambda_2} \frac{\lambda_2}{\lambda_2+\lambda_3} \frac{\lambda_3}{\lambda_3+\lambda_4} + \frac{\lambda_2}{\lambda_1+\lambda_2} \frac{\lambda_1}{\lambda_1+\lambda_4} \frac{\lambda_3}{\lambda_3+\lambda_4}\right) \\ p_7 &= \frac{1}{4} \frac{\lambda_1}{\lambda_1+\lambda_2} \\ p_8 &= \frac{1}{4} \frac{\lambda_2}{\lambda_1+\lambda_2} \end{aligned}$$

(v) The vector of expected waiting times Ψ is:

$$\Psi = \left(\frac{1}{\lambda_1}, \frac{1}{\lambda_2}, \frac{1}{\lambda_3+\lambda_4}, \frac{1}{\lambda_3}, \frac{1}{\lambda_4}, \frac{1}{\lambda_1+\lambda_2}, \frac{1}{\lambda_2+\lambda_3}, \frac{1}{\lambda_1+\lambda_4}\right)$$

(vi) The limiting process probabilities are calculated as:

$$\Pi_i = \frac{p_i\Psi_i}{\sum_{j=1}^{8} p_j\Psi_j}, i = 1,\ldots,8$$

(vii) If we assume that we have identical rates $\lambda_1 = \lambda_2 = \lambda_3 = \lambda_4 = 1$, we get

$$\begin{aligned} p &= \frac{1}{4}\left(\frac{1}{4}, \frac{1}{4}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, 1, \frac{1}{2}, \frac{1}{2}\right), \\ \Psi &= \frac{1}{2}(2,2,1,2,2,1,1,1), \text{ and} \\ \Pi &= \frac{1}{11}(1,1,1,2,2,2,1,1). \end{aligned}$$

6.5 Future Directions

In this paper we have demonstrated how the aggregation principle from [Sch84] can be used to analyze periodic and irreducible M–TDFD's with finite reachability sets. Especially for large models, this approach is very helpful to efficiently determine stationary probabilities, expected waiting times, and limiting process probabilities. So far, we have used this approach only to analyze periodic and irreducible M–TDFD's with finite reachability sets. Future work can be directed into two directions:

- Analysis of M–TDFD's with infinite reachability sets: Many real systems behave like queueing systems or queueing networks with finite or infinite queue lengths and tend to be periodic and irreducible. M–TDFD's representing such systems might be candidates to be analyzed in a manner similiar to the one described in this paper.

- Analysis of TDFD's with arbitrary transition times: The main idea in [Sch84] was the application of the aggregation principle to queueing systems and networks with arbitrary service and inter–arrival times, approximated through mixtures of Erlang distributions. We might be capable to do a computationally efficient analysis of TDFD's where transition times are modeled as mixtures of Erlang distributions, using the aggregation principle in its original context.

Finally, there exist several types of real systems that are good candidates to be correctly modeled and analyzed through (periodic and irreducible) M–TDFD's, while currently still being modeled and

analyzed through Timed Petri Nets. Examples for these systems are communication protocols (e. g., [MAT+77], [MB83], [Wal83]) and complex computer systems (e. g., [Zub80]). Another type of system that might work quite well are general Producer/Consumer systems or networks of these, e. g., multi–stage production systems (e. g., [Hil90]).

However, many systems modeled through Data Flow Diagrams, e. g., the case study of an elevator system in [You89], the cruise control system, the bottle–filling system, the pocket–sized logic analyser, and the defect inspection system, all in [WM85b], probably would have non–Markovian transition times. For models like these, an approximation of the real time behavior through mixtures of Erlang distributions might be possible and an analysis based on the aggregation principle should be preferable to results gained from simulation runs based on the TDFD.

Acknowledgements

Symanzik's research was partially supported by a German "DAAD–Doktorandenstipendium aus Mitteln des zweiten Hochschulsonderprogramms". The author wishes to thank Herbert T. David and Albert L. Baker for many valuable suggestions and references, including referral to the Ph.D. dissertation of Hoon–Shik Woo.

7 GENERAL SUMMARY

7.1 Discussion of Results

The papers that make up this Ph.D. dissertation establish a firmer theoretical foundation for Formalized Data Flow Diagrams (FDFD's) and extend FDFD's to include a general notation of time. The main results of each of the five papers are, briefly stated:

- PFF-RDFD's are Turing equivalent.
- Stores, persistent flows, tests for empty flows, and infinite domains are not essential for FDFD's.
- Subclasses of FDFD's are equivalent to known subclasses of FIFO Petri Nets, immediately furnishing the decidability results for subclasses of FIFO Petri Nets to the corresponding subclasses of FDFD's.
- A general stochastic model of time for FDFD's (called Timed Data Flow Diagrams — TDFD's) is defined, allowing not only a description of the relative likelihoods of various execution times, but also descriptions of the possible joint firing behavior of transitions.
- An aggregation principle can be used for an efficient stochastic analysis of periodic TDFD's with Markovian transition times.

The relation among computational models defined in this Ph.D. dissertation and other well-known models are summarized in Figure 7.1. The notation $X \sim Y$ means that models X and Y are computationally equivalent. The notation $X \longrightarrow Y$ is used to indicate that model X can be simulated by model Y. A model X that is at the top end of a connecting line | is more powerful than a model Y at the bottom end of such a line. Models that have been introduced in this Ph.D. dissertation are labeled with **bold** letters while those models that were previously known are labeled with *italic* letters. Similarly, for simulations that have been introduced in this Ph.D. dissertation, thick arrows have been used while thin arrows have been used for previously known simulations.

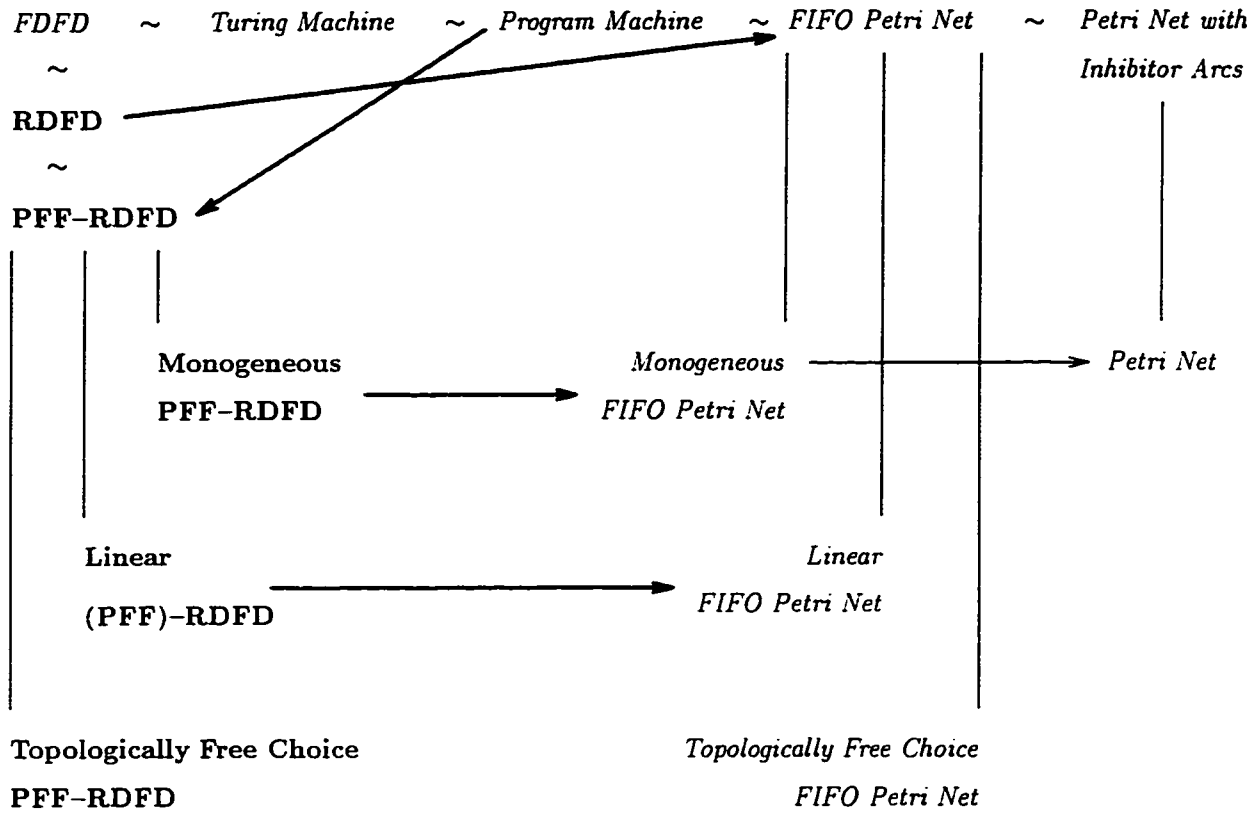

Figure 7.1: Relation among Computational Models.

In Figure 7.2 the relation between timed computational models and stochastic processes has been summarized. The same symbolism as in Figure 7.1 has been used with the exception that the notation $X \longrightarrow Y$ now is used to indicate that model X is mapped to the (stochastic) model Y.

7.2 Further Research

This research leaves several interesting questions unanswered. It was not intended to solve any of the following problems for this dissertation. In addition to the proposals for future research given at the end of each chapter (paper), we want to highlight a few topics that are of particular interest for further investigations.

In Chapter 4, it is shown that some decidability problems for subclasses of FDFD's are solvable by solving the corresponding problem for FIFO Petri Nets. Moreover, we know criteria from the literature to determine whether a given FIFO Petri Net belongs to any of these subclasses. There also exist algorithms that solve some of the decidability problems for FIFO Petri Nets. On the other hand,

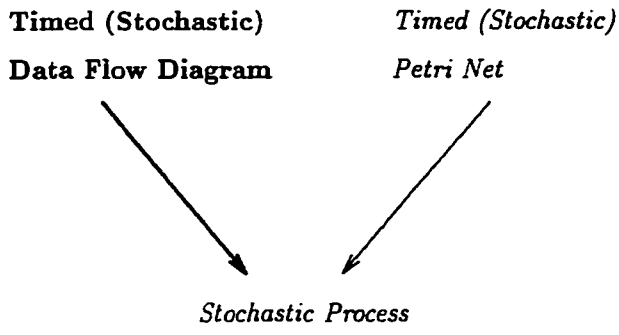

Figure 7.2: Relation among Timed Computational Models.

we have an operational (i. e., executable) semantics for FDFD's ([WBL93]). It should be possible to implement efficient algorithms that (i) check whether a given FDFD belongs to any known subclass of FDFD's, and (ii) if so, that solve the interesting decidability problems. Hopefully, an implementation does not require the formal mapping from an FDFD onto a FIFO Petri Net.

There might be additional subclasses of FDFD's that do not map to FIFO Petri Nets, but there might be decidability problems that are solvable for these subclasses. For example, "conflicts", i. e., activities competing for the same resources, occur only in few FDFD's. There might be some decidability problems that can be solved for FDFD's without conflicts.

For TDFD's, all calculations done so far have been done manually. It is desirable to implement algorithms (e. g., the one introduced in Chapter 6) that make use of the TDFD representation and evaluate and analyze the model. One might wish to obtain software that is capable of doing an analytical analysis of the TDFD as well as simulations if an analytical approach is impossible.

A lot of work has been done to analyze Timed Petri Nets. Many of those methods and algorithms could be easily adapted for some subclasses of TDFD's. It is desirable to classify such subclasses. Another question deals with the adequacy of the model in comparison to the real distributed system. How well do system and model match? If they match well, is the model still analytically analyzable or is simulation the only way to receive interesting answers?

So far, we have completely neglected decidability problems for TDFD's. Whenever the reachability set of a TDFD is identical to the reachability set of its underlying FDFD, it is sufficient to consider the FDFD only. Differences occur if the reachability sets differ. This can happen because of 0–probabilities to select a possible transition and because of distributions of time that do not cover the entire real coordinate axxis but only a limited subset. In particular, Dirac distributions that represent a deterministic time for each transition can result in a completely different behavior of the system.

States that have been reachable for the FDFD may not be reachable for the TDFD. One might wish to implement software that is capable to answer decidability problems for subclasses of TDFD's.

And finally, we have to consider questions about the "expressive convenience" of FDFD's/TDFD's in comparison to (Timed) Petri Nets. What kind of system analyst is more likely to use what type of model for a real system. What are the reasons for selecting a particular model? What kind of real system is easy to model within one model but hardly to model within the other? These questions will hopefully be answered when a larger number of system analysts has gained further experience with both types of computational models.

BIBLIOGRAPHY

[AD88] E. Abdurachman and H.T. David. Cesaro Limits of Marked Point Processes on the Line. *Communication in Statistics — Stochastic Models*, 4(1):77–98, 1988.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison–Wesley, Reading, Massachusetts, 1974.

[BB93] J.P. Bansler and K. Bødker. A Reappraisal of Structured Analysis: Design in an Organizational Context. *ACM Transactions on Information Systems*, 11(2):165–193, 1993.

[BR76] G. Berthelot and G. Roucairol. Reduction of Petri–Nets. In A. Mazurkiewicz, editor, *Lecture Notes in Computer Science Vol. 45: Mathematical Foundations of Computer Science 1976: Proceedings, 5th Symposium, Gdańsk, September 1976*, pages 202–207, Springer–Verlag, Berlin, Heidelberg, 1976.

[BR90] I.I. Bestuzheva and V.V. Rudnev. Timed Petri Nets: Classification and Comparative Analysis. *Automation and Remote Control, Pt. 1*, 51(10):1303–1318, 1990.

[CB94] D.L. Coleman and A.L. Baker. Synthesizing Structured Analysis and Object–Oriented Specifications. Technical Report 94-04, Iowa State University, Department of Computer Science, 226 Atanasoff Hall, Ames, Iowa 50011, March 1994. Available by anonymous ftp from ftp.cs.iastate.edu or by e-mail from almanac@cs.iastate.edu.

[CF87] A. Choquet and A. Finkel. *Simulation of Linear FIFO Nets by a new Class of Petri Nets*. Universite de Paris-Sud, Centre d'Orsay, Laboratoire de Recherche en Informatique, Bat. 490, 91405 Orsay (France), Rapport de Recherche No. 324, Jan 1987.

[CGL94] G. Ciardo, R. German, and C. Lindemann. A Charcterization of the Stochastic Process Underlying a Stochastic Petri Net. *IEEE Transactions on Software Engineering*, 20(7):506–515, 1994.

[Chi85] G. Chiola. A Software Package for the Analysis of Generalized Stochastic Petri Net Models. In *International Workshop on Timed Petri Nets, Torino, Italy, July 1985*, pages 136–143, 1985.

[Col91] D.L. Coleman. *Formalized Structured Analysis Specifications.* PhD Thesis, Iowa State University, Ames, Iowa, 50011, 1991.

[Cum85] A. Cumani. ESP — A Package for the Evaluation of Stochastic Petri Nets with Phase–Type Distributed Transition Times. In *International Workshop on Timed Petri Nets, Torino, Italy, July 1985*, pages 144–151, 1985.

[DeM78] T. DeMarco. *Structured Analysis and System Specification.* Yourdon, Inc., New York, New York, 1978.

[Fan92] J. Fanchon. A FIFO–Net Model for Processes with Asynchronous Communication. In G. Rozenberg, editor, *Lecture Notes in Computer Science Vol. 609: Advances in Petri Nets 1992*, pages 152–178, Springer–Verlag, Berlin, Heidelberg, 1992.

[FC88] A. Finkel and A. Choquet. FIFO Nets Without Order Deadlock. *Acta Informatica*, 25(1):15–36, 1988.

[Fin84] A. Finkel. Petri Nets and Monogeneous FIFO Nets. *Bulletin of the European Association of Theoretical Computer Science*, 23:28–31, 1984.

[Fin86] A. Finkel. *Structuration des Systemes de Transitions — Applications au Controle du Parallelisme par Files FIFO.* These Science, Universite de Paris–Sud, Centre d'Orsay, 1986.

[FM82] A. Finkel and G. Memmi. FIFO Nets: A New Model of Parallel Computation. In A.B. Cremers and H.P. Kriegel, editors, *Lecture Notes in Computer Science Vol. 145: Theoretical Computer Science: 6th GI–Conference, Dortmund, January 1983*, pages 111–121, Springer–Verlag, Berlin, Heidelberg, 1982.

[FR85] M.P. Flé and G. Roucairol. Fair Serializability of Iterated Transactions Using FIFO-Nets. In G. Rozenberg, editor, *Lecture Notes in Computer Science Vol. 188: Advances in Petri Nets 1984*, pages 154–168, Springer–Verlag, Berlin, Heidelberg, 1985.

[FR88] A. Finkel and L. Rosier. A Survey on the Decidability Questions for Classes of FIFO Nets. In G. Rozenberg, editor, *Lecture Notes in Computer Science Vol. 340: Advances in Petri Nets 1988*, pages 106–132, Springer–Verlag, Berlin, Heidelberg, 1988.

[GHG+93] J.V. Guttag, J.J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer–Verlag, New York, 1993.

[GM95] R. German and J. Mitzlaff. Transient Analysis of Deterministic and Stochastic Petri Nets with TimeNET. In H. Beilner and F. Bause, editors, *Lecture Notes in Computer Science Vol. 977: Quantitative Evaluation of Computing and Communication Systems*, pages 209–223, Springer–Verlag, Berlin, Heidelberg, 1995.

[God82] H.P. Godbersen. On the Problem of Time in Nets. In C. Girault and W. Reisig, editors, *Informatik-Fachberichte Vol. 52: Application and Theory of Petri Nets*, pages 23–30, Springer–Verlag, Berlin, Heidelberg, 1982.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Har92] D. Harel. Biting the Silver Bullet. *Computer*, 21(1):8–20, January 1992.

[Har96] D. Harel. Executable Object Modeling with Statecharts. In *Proceedings of the 18th International Conference on Software Engineering*, pages 246–257. IEEE Computer Society Press, January 1996.

[Hil90] H.P. Hillion. Timed Petri Nets and Application to Multi–Stage Production Systems. In G. Rozenberg, editor, *Lecture Notes in Computer Science Vol. 424: Advances in Petri Nets 1989*, pages 281–305, Springer–Verlag, Berlin, Heidelberg, 1990.

[HT91] W. Henderson and P.G. Taylor. Embedded Processes in Stochastic Petri Nets. *IEEE Transactions on Software Engineering*, 17(2):108–116, 1991.

[IM76] D.L. Isaacson and R.W. Madsen. *Markov Chains. Theory and Applications.* Wiley, New York, London, Sydney, Toronto, 1976.

[Jen80] K. Jensen. A Method to Compare the Descriptive Power of Different Types of Petri Nets. In P. Dembiński, editor, *Lecture Notes in Computer Science Vol. 88: Mathematical Foundations of Computer Science 1980: Proceedings of the 9th Symposium Held in Rydzyna, Poland, September 1980*, pages 348–361, Springer–Verlag, Berlin, Heidelberg, 1980.

[Jon86] C.B. Jones. *Systematic Software Development Using VDM.* International Series in Computer Science. Prentice–Hall, Englewood Cliffs, New Jersey, 1986.

[KM82] T. Kasai and R.E. Miller. Homomorphisms between Models of Parallel Computation. *Journal of Computer and System Sciences*, 25:285–331, 1982.

[KSK76] J.G. Kemeny, J.L. Snell, and A.W. Knapp. *Denumerable Markov Chains (Second Edition)*. Springer–Verlag, New York, 1976.

[LP81] H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation*. Prentice–Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[LWBL96] G.T. Leavens, T. Wahls, A.L. Baker, and K. Lyle. An Operational Semantics of Firing Rules for Structured Analysis Style Data Flow Diagrams. Technical Report 93–28d, Iowa State University, Department of Computer Science, 226 Atanasoff Hall, Ames, Iowa 50011, December 1993, revised, July 1996. Available by anonymous ftp from ftp.cs.iastate.edu or by e–mail from almanac@cs.iastate.edu.

[Mat70] J.V. Matijasevic. Enumerable Sets are Diophantine. *Soviet Mathematics, Doklady*, 11(2):354–358, 1970.

[MAT+77] M. Mori, T. Araki, K. Taniguchi, N. Tokura, and T. Kasami. Some Decision Problems for Time Petri Nets and Applications to the Verification of Communication Protocols. *Transactions of the Institute of Electronics and Communication Engineers of Japan, Section E (English)*, 60(10):598–599, 1977.

[MB83] M. Menasche and B. Berthomieu. Time Petri Nets for Analyzing and Verifying Time Dependent Communication Protocols. In H. Rudin and C.H. West, editors, *Protocol Specification, Testing, and Verification, III*, pages 161–172. Elsevier (North–Holland), 1983.

[MBB+85] M.A. Marsan, G. Balbo, A. Bobbio, G. Chiola, G. Conte, and A. Cumani. On Petri Nets with Stochastic Timing. In *International Workshop on Timed Petri Nets, Torino, Italy, July 1985*, pages 80–87, 1985.

[MC87] M.A. Marsan and G. Chiola. On Petri Nets with Determinstic and Exponentially Distributed Firing Times. In G. Rozenberg, editor, *Lecture Notes in Computer Science Vol. 266: Advances in Petri Nets 1987*, pages 132–145. Springer–Verlag, Berlin, Heidelberg, 1987.

[Men85] M. Menasche. PAREDE: An Automated Tool for the Analysis of Time(d) Petri Nets. In *International Workshop on Timed Petri Nets, Torino, Italy, July 1985*, pages 162–169, 1985.

[MF85] G. Memmi and A. Finkel. An Introduction to FIFO Nets — Monogeneous Nets: A Subclass of FIFO Nets. *Theoretical Computer Science*, 35(2–3):191–214, 1985.

[Min67] M.L. Minsky. *Computation: Finite and Infinite Machines.* Prentice–Hall, Inc., Englewood Cliffs, New Jersey, 1967.

[MM81] R. Martin and G. Memmi. Specification and Validation of Sequential Processes Communicating by FIFO Channels. *I.E.E. Conference Publication No. 198: Fourth International Conference on Software Engineering for Telecommunication Switching Systems, Warwick, July 1981*, pages 54–57, 1981.

[Nat96] National Academy of Sciences. *Statistical Software Engineering.* National Academy Press, Washington, D.C., 1996.

[Pat64] R.L. Patterson. Markov Processes Occurring in the Theory of Traffic Flow through an N–Stage Stochastic Service System. *Journal of Industrial Engineering*, 15:188–193, 1964.

[Pet81] J.L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice–Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[PS82] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice–Hall, Inc., Englewood Cliffs, New Jersey, 1982.

[Rou87] G. Roucairol. FIFO–Nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Lecture Notes in Computer Science Vol. 254: Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I. Proceedings of an Advanced Course, Bad Honnef, September 1986*, pages 436–459. Springer–Verlag, Berlin, Heidelberg, 1987.

[RP84] R.R. Razouk and C.V. Phelps. Performance Analysis using Timed Petri Nets. In R.M. Keller, editor, *Proceedings of the 1984 International Conference on Parallel Processing*, pages 126–128, IEEE Computer Society Press, Silver Spring, Maryland, 1984.

[SB96a] J. Symanzik and A.L. Baker. Formalized Data Flow Diagrams and Their Relation to Other Computational Models. Technical Report 96–20. Iowa State University, Department of Computer Science, 226 Atanasoff Hall, Ames, Iowa 50011, December 1996. Available by anonymous ftp from ftp.cs.iastate.edu or by e–mail from almanac@cs.iastate.edu.

[SB96b] J. Symanzik and A.L. Baker. Non–Atomic Components of Data Flow Diagrams: Stores, Persistent Flows, and Tests for Empty Flows. Technical Report 96–21, Iowa State University, Department of Computer Science, 226 Atanasoff Hall, Ames, Iowa 50011, December 1996. Available by anonymous ftp from ftp.cs.iastate.edu or by e–mail from almanac@cs.iastate.edu.

[SB96c] J. Symanzik and A.L. Baker. Subclasses of Formalized Data Flow Diagrams: Monogeneous, Linear, and Topologically Free Choice RDFD's. Technical Report 96–22, Iowa State University, Department of Computer Science, 226 Atanasoff Hall, Ames, Iowa 50011, December 1996. Available by anonymous ftp from ftp.cs.iastate.edu or by e–mail from almanac@cs.iastate.edu.

[SB96d] J. Symanzik and A.L. Baker. Timed Data Flow Diagrams. Technical Report 96–23, Iowa State University. Department of Computer Science, 226 Atanasoff Hall, Ames, Iowa 50011, December 1996. Available by anonymous ftp from ftp.cs.iastate.edu or by e–mail from almanac@cs.iastate.edu.

[Sch84] R. Schassberger. An Aggregation Principle for Computing Invariant Probability Vectors in Semi–Markovian Models. In G. Iazeolla, P.J. Courtois, and A. Hordijk, editors, *Mathematical Computer Performance and Reliability*, pages 259–273, Elsevier (North–Holland), Amsterdam, 1984.

[Sta83] P.H. Starke. Monogenous FIFO–Nets and Petri–Nets are Equivalent. *Bulletin of the European Association of Theoretical Computer Science*, 21:68–77, 1983.

[TP89] T.H. Tse and L. Pong. Towards a Formal Foundation for DeMarco Data Flow Diagrams. *The Computer Journal*, 32(1):1–12, February 1989.

[VVN81] R. Valk and G. Vidal-Naquet. Petri Nets and Regular Languages. *Journal of Computer and System Sciences*, 23:299–325, 1981.

[Wah95] T. Wahls. *On the Execution of High Level Formal Specifications*. PhD Thesis, Iowa State University, Ames, Iowa, 50011, 1995.

[Wal83] B. Walter. Timed Petri–Nets for Modelling and Analyzing Protocols with Real–Time Characteristics. In H. Rudin and C.H. West, editors, *Protocol Specification, Testing, and Verification, III*, pages 161–172. Elsevier (North–Holland), 1983.

[War86] P.T. Ward. The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing. *IEEE Transactions on Software Engineering*, SE–12(2):198–210, 1986.

[WBL93] T. Wahls, A.L. Baker, and G.T. Leavens. An Executable Semantics for a Formalized Data Flow Diagram Specification Language. Technical Report 93–27, Iowa State University, Department of Computer Science, 226 Atanasoff Hall, Ames, Iowa 50011, November 1993. Available by anonymous ftp from ftp.cs.iastate.edu or by e–mail from almanac@cs.iastate.edu.

[WBL94] T. Wahls, A.L. Baker, and G.T. Leavens. The Direct Execution of SPECS–C++: A Model–Based Specification Language for C++ Classes. Technical Report 94–02b, Iowa State University, Department of Computer Science, 226 Atanasoff Hall, Ames, Iowa 50011, February 1994, revised. November 1994. Available by anonymous ftp from ftp.cs.iastate.edu or by e–mail from almanac@cs.iastate.edu.

[WM85a] P.T. Ward and S.J. Mellor. *Structured Development for Real–Time Systems*, Volume 1: Introduction and Tools. Yourdon, Inc., New York, New York, 1985.

[WM85b] P.T. Ward and S.J. Mellor. *Structured Development for Real–Time Systems*, Volume 2: Essential Modeling Techniques. Yourdon, Inc., New York, New York, 1985.

[Woo93] H.-S. Woo. *On Deterministic and Markovian Production Systems.* PhD Thesis, Iowa State University, Ames, Iowa, 50011, 1993.

[You89] E. Yourdon. *Modern Structured Analysis.* Yourdon Press Computing Series. Prentice–Hall, Englewood Cliffs, New Jersey, 1989.

[Yua86] C.Y. Yuan. Process Periods and System Reconstruction. In G. Rozenberg, editor, *Lecture Notes in Computer Science Vol. 222: Advances in Petri Nets 1985*, pages 122–141, Springer–Verlag, Berlin, Heidelberg, 1986.

[Zub80] W.M. Zuberek. Timed Petri Nets and Preliminary Performance Evaluation. In *IEEE Proceedings of the 7th Annual Symposium on Computer Architecture, May 1980, La Baule, France*, pages 88–96, 1980.

ACKNOWLEDGEMENTS

First of all, I am happy and proud to acknowledge that this Ph.D. dissertation was supported by a German "DAAD–Doktorandenstipendium aus Mitteln des zweiten Hochschulsonderprogramms".

Then, I would like to express my deepest gratitude to my two co–major professors Albert L. Baker and Herbert T. David for their encouragement, enthusiasm, inspiration, and untiring patience throughout the course of this research. My appreciation also goes to all current and former members of my Ph.D. committee, professors Sharon Filipowski, Kenneth J. Koehler, James H. Oliver, Donald L. Pigozzi, and Johnny S. Wong, for their help, time, and support in serving on my program of study committee.

I am grateful to Alain Finkel for providing me with a copy of his Ph.D. dissertation on FIFO Petri Nets, an important source for my dissertation otherwise unavailable to me. My appreciation also goes to those people at the Computer Science Department, University of Aarhus, Denmark, for maintaining the Petri Net bibliography database on the WWW at the URL `http://www.daimi.aau.dk/~petrinet/bibl/pnbibl.html`, a very important source to locate references relevant for my Ph.D. dissertation. I also want to thank everyone who helped me with discussions, references, LaTeX problems, and many other details required to finish this Ph.D. dissertation.

And last, but not least, I would like to say "Herzlichen Dank" to my parents, Gerd and Gisela Symanzik, who gave me the best possible support and encouragement during all these years of my academic career.

BIOGRAPHICAL SKETCH

Jürgen Symanzik was born November 11, 1965 in Marl, Germany. He received the degree Diplom–Statistiker (M.S. in Statistics) in 1991 and the degree Diplom–Informatiker (M.S. in Computer Science) in 1992, both from the Universität Dortmund, Dortmund, Germany.

From 1993 to 1995 he was awarded a "DAAD–Doktorandenstipendium aus Mitteln des zweiten Hochschulsonderprogramms" (a three year Ph.D. research fellowship from the German Academic Exchange Service). He has served as a Research Assistant and a Teaching Assistant in the Department of Statistics and in the Department of Computer Science at the Universität Dortmund from 1989 to 1992. He has also served as a Research Assistant in the Department of Statistics and in the Geographic Information System (GIS) Facility at Iowa State University from 1993 to 1996.

His current research interests are in Computational Models (e. g., Formalized Data Flow Diagrams, Timed Data Flow Diagrams, FIFO Petri Nets) and in Dynamic Statistical Graphics with applications in Geographic Information Systems and Virtual Reality. An up–to–date overview of his professional publications, presentations, and research projects can be found on the WWW at the URL `http://www.public.iastate.edu/~symanzik/papers/papers.html`.